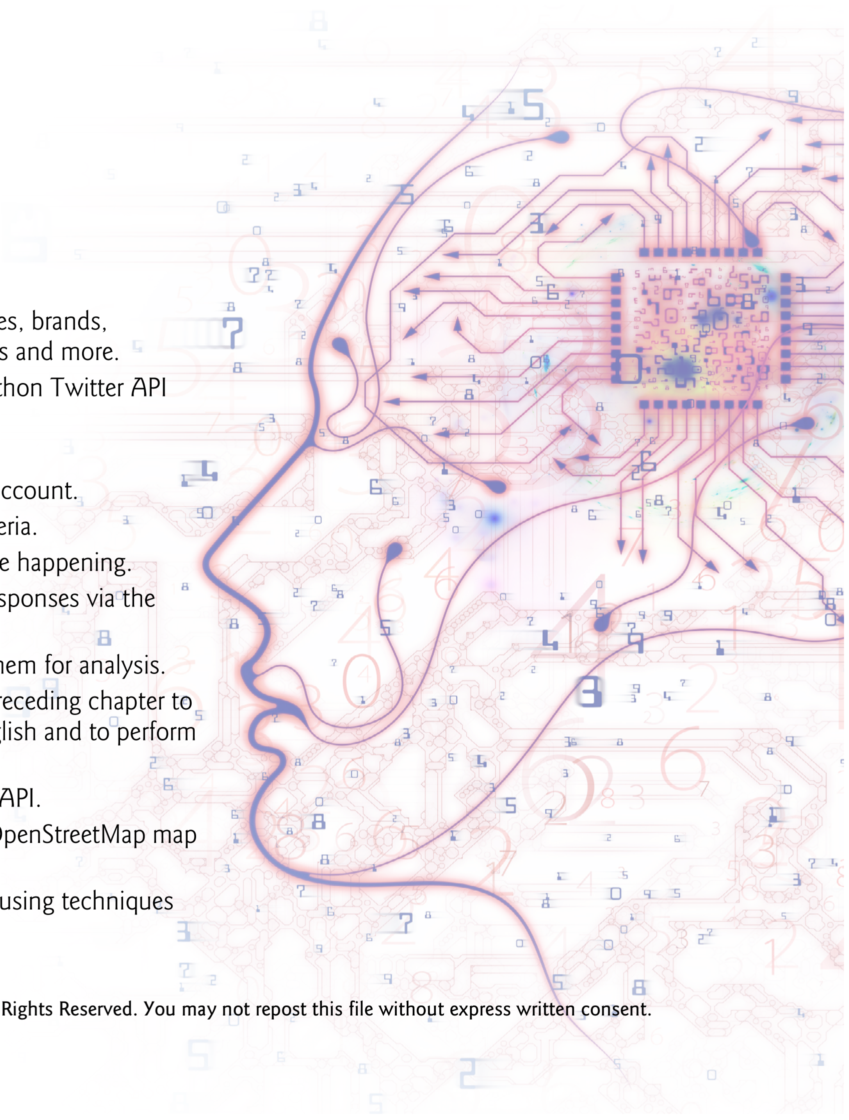
12

Data Mining Twitter



Objectives

In this chapter, you'll:

- Understand Twitter's impact on businesses, brands, reputation, sentiment analysis, predictions and more.
- Use Tweepy, one of the most popular Python Twitter API clients for data mining Twitter.
- Use various Twitter v2 API methods.
- Get information about a specific Twitter account.
- Search for past tweets that meet your criteria.
- Sample the stream of live tweets as they're happening.
- Request additional metadata in Twitter responses via the Twitter v2 API's expansions and fields.
- Clean and preprocess tweets to prepare them for analysis.
- Use NLP techniques you learned in the preceding chapter to translate foreign language tweets into English and to perform sentiment analysis on tweets.
- Spot trends with the Twitter v1.1 Trends API.
- Map tweets using the folium library and OpenStreetMap map tiles.
- Understand various ways to store tweets using techniques discussed throughout this book.

Outline

12.1 Introduction
12.2 Overview of the Twitter APIs
12.3 Creating a Twitter Developer Account
12.4 Getting Twitter Credentials—Creating an App
12.5 What's in a Twitter API Response?
12.6 Installing Tweepy, `geopy`, `folium` and `deep-translator`
12.7 Authenticating with Twitter Via Tweepy to Access Twitter v2 APIs
12.8 Getting Information About a Twitter Account
12.9 Intro to Tweepy `Paginator`s: Getting More Than One Page of Results
12.9.1 Determining an Account's Followers
12.9.2 Determining Whom an Account Follows
12.9.3 Getting a User's Recent Tweets
12.10 Searching Recent Tweets; Intro to Twitter v2 API Search Operators
12.11 Spotting Trending Topics
12.11.1 Places with Trending Topics
12.11.2 Getting a List of Trending Topics
12.11.3 Create a Word Cloud from Trending Topics
12.12 Cleaning/Preprocessing Tweets for Analysis
12.13 Twitter Streaming API
12.13.1 Creating a Subclass of `StreamingClient`
12.13.2 Initiating Stream Processing
12.14 Tweet Sentiment Analysis
12.15 Geocoding and Mapping
12.15.1 Getting and Mapping the Tweets
12.15.2 Utility Functions in `tweetutilities.py`
12.15.3 Class `LocationListener`
12.16 Storing Tweets
12.17 Twitter and Time Series
12.18 Wrap-Up

12.1 Introduction

We're always trying to predict the future. Will it rain at our upcoming picnic? Will the stock market or individual securities go up or down? When and by how much? How will people vote in the next election? What's the chance that a new oil exploration venture will strike oil, and if so, how much would it likely produce? Will a baseball team win more games if it changes its batting philosophy to "swing for the fences?" How much customer traffic does an airline anticipate over the next many months? And hence how should the company buy oil commodity futures to guarantee that it will have the supply it needs and hopefully at a minimal cost? What track is a hurricane likely to take, and how powerful will it become (category 1, 2, 3, 4 or 5)? That kind of information is crucial to emergency preparedness efforts. Is a financial transaction likely to be fraudulent? Will a mortgage default? Is a disease likely to spread rapidly, and if so, to what geographic area?

Prediction is a challenging and often costly process, but the rewards can be significant. Using the technologies in this and the upcoming chapters, you'll see how AI, often in concert with big data, is rapidly improving prediction capabilities.

Data Mining

This chapter focuses on data mining Twitter, looking for the sentiment in tweets. **Data mining** is the process of searching through extensive collections of data, often big data, to find insights that can be valuable to individuals and organizations. The sentiment that you data mine from tweets could help predict the results of an election, the revenues a new movie is likely to generate and the success of a company's marketing campaign. It could also help companies spot weaknesses in competitors' product offerings.

Twitter v2 (version 2) Web Service APIs

You'll interact with the Twitter v2 (version 2) web service APIs. You'll use search criteria to locate tweets in the enormous base of past tweets. You'll tap into Twitter's live tweet stream to receive new tweets as they happen. You'll locate worldwide and specific locations' trending topics. You'll find that much of what you learned in the NLP chapter will be useful in building Twitter applications.

As you've done throughout this book, you'll use powerful libraries to perform significant tasks with just a few lines of code. This is why Python and its robust open-source community are appealing.

The Twitterverse

Twitter has displaced the major news organizations as the first source for newsworthy events—in this sense, Twitter is a classic **disruptive technology**. Most Twitter posts are public and happen in real time as events unfold globally. People speak frankly about any subject and tweet about their personal and business lives. They comment on the social, entertainment and political scenes and whatever else comes to mind. With their mobile phones, they take and post photos and videos of events as they happen. You'll hear the terms **Twitterverse** and **Twittersphere** to mean the hundreds of millions of users who have anything to do with sending, receiving and analyzing tweets.

What Is Twitter?

Twitter was founded in 2006 as a **microblogging** company and today is one of the most popular sites on the Internet. Its concept is simple. People write short messages called *tweets*. Initially, these were limited to 140 characters but are now limited to 280 characters. Anyone can generally choose to follow the tweets of anyone else. This differs from the closed, tight communities on social media platforms such as Meta (formerly called Facebook), LinkedIn and many others, where "following relationships" must be reciprocal.

Twitter Statistics

Twitter has hundreds of millions of users. Based on the following Twitter Statistics page

```
https://www.internetlivestats.com/twitter-statistics/
```

we calculated in August 2022 that there is an average of 10,000+ tweets per second, resulting in about 880 million tweets per day. Searching online for "Internet statistics" and "Twitter statistics" will help you put these numbers in perspective. Some "tweeters" have more than 100 million followers. Dedicated tweeters generally post several per day to keep their followers engaged. Tweeters with the largest followings are typically entertainers and politicians. Developers can tap into the live stream of tweets as they're happening. This has been likened to "drinking from a fire hose" because the tweets flow to you so quickly.

Twitter and Big Data

Twitter has become a favorite big data source for researchers and business people worldwide. Developers have free access to a small portion of the more recent tweets, subject to tweet caps by their account type.[1] Twitter offers paid access to much larger portions of the all-time tweets database.

1. `https://developer.twitter.com/en/docs/twitter-api/tweet-caps`. Accessed August 25, 2022.

Cautions

You can't always trust everything you read on the Internet, and tweets are no exception. For example, people might use false information (i.e., "fake news") to manipulate financial markets or influence political elections. Hedge funds often trade securities based partly on the tweet streams they follow, but they're cautious. That's one of the challenges of building business-critical or mission-critical systems based on social media content.

We use web services extensively throughout the book. Internet connections can be lost, services can change, and some services are not available in all countries. This is the real world of cloud-based programming. We cannot program with the same reliability as desktop apps when using web services.

12.2 Overview of the Twitter APIs

Twitter's APIs are cloud-based web services, so an Internet connection is required to execute the code in this chapter. **Web services** are methods you call in the cloud, as you'll do with the Twitter APIs in this chapter, the IBM Watson APIs in the next chapter and other APIs you'll use as computing becomes more cloud-based. Each API method has a web service **endpoint**, represented by a URL that's used to invoke that method over the Internet.

The Twitter v2 APIs include many categories of functionality, some free and some paid. Most have **rate limits** that restrict the number of times you can use them in 15-minute intervals. In this chapter, you'll use the **Tweepy library** to invoke methods from the following Twitter API categories:

- **Users API**—Access information about Twitter user accounts.
- **Tweets API**—Search through past tweets, access tweet streams to tap into tweets happening now and more.
- **Trends API** (from the Twitter v1.1 APIs)—Find locations of trending topics and get lists of trending topics by location.

For additional Twitter API categories and subcategories and their methods, see:

```
https://developer.twitter.com/en/docs/api-reference-index
```

Rate Limits: A Word of Caution

Twitter expects developers to use its services responsibly. Each Twitter API method has a **rate limit**, which is the maximum number of requests (i.e., calls to that method) you can make during a 15-minute window. Twitter may block you from using its APIs if you continue to call a given API method after its rate limit has been reached.

Before using any API method, read its documentation and understand its rate limits.[2] As you'll see, Tweepy can wait when it encounters rate limits to prevent you from exceeding Twitter's rate-limit restrictions. Some methods list both user rate limits and app rate limits. All of this chapter's examples use app rate limits. User rate limits are for apps that enable individual users to log into Twitter, such as smartphone apps that interact with Twitter on your behalf.

For details on rate limiting, see

```
https://developer.twitter.com/en/docs/rate-limits
```

2. Keep in mind that Twitter could change these limits.

For specific rate limits on individual API methods, see

`https://developer.twitter.com/en/docs/twitter-api/rate-limits`

and each Twitter API method's documentation.

Other Restrictions

Twitter's free APIs are a goldmine for data mining. You'll be amazed at the applications you can build and how these will help you improve your personal and career endeavors. **However, your developer account could be terminated if you do not follow Twitter's rules and regulations. You should carefully read Twitter's Terms of Service**

`https://twitter.com/tos`

and the documents it links to.

You'll see later in this chapter that you can search tweets only for the last seven days and get only a limited number of tweets using the free Twitter APIs. Some books and articles say you can get around those limits by scraping tweets directly from `twitter.com`. However, the Terms of Service explicitly say that "**scraping the Services without the prior consent of Twitter is expressly prohibited.**"

12.3 Creating a Twitter Developer Account

Twitter requires you to apply for a developer account to be able to use their APIs. Go to

`https://developer.twitter.com/`

and click the **Sign up** button. If you do not already have a Twitter account, you must register for one as part of the developer-account sign-up process.

Twitter Developer Account Levels

Twitter reviews every developer-account application, and approval is not guaranteed. If you are approved, your developer account will have one of three levels, which Twitter describes as follows:[3]

- **Essentials**—"The best way to get started quickly, test, and build across all endpoints."
- **Elevated**— "More access for solutions that are beginning to experience growth or who prefer to work with multiple App environments."
- **Academic Research**—"Access to public data on nearly any topic to advance research objectives of Master's students, doctoral candidates, post-docs, and faculty at an academic institution or university."

Some Twitter v2 APIs are accessible only to Elevated-level and higher accounts. For each API, the Twitter documentation specifies the minimum account level and the rate-limit differences between levels, if any.

Choosing a Developer Account Application Type

There are separate developer applications for **Professional**, **Hobbyist**, and **Academic Research** use. You should choose the type most appropriate for your use case. For this

3. `https://developer.twitter.com/en/products/twitter-api`. Accessed August 27, 2022.

chapter's examples, you can choose **Hobbyist** then **Exploring the API**. You may be asked to apply for an **Elevated application**. If so, click **Get started**, then:

1. On the **Basic info** tab, fill in the form with your information and click **Next**.
2. On the **Intended use** tab, describe how you intend to use the APIs.
3. Answer the other questions provided. For this chapter's examples, you will not use the tweet, retweet, like, follow or direct message functionality; will not display tweets or aggregate data about Twitter content outside of Twitter; and will not make Twitter content available to a government entity.
4. Click **Next** to review your answers, then click **Next** again.
5. Carefully read and agree to Twitter's **Developer agreement & policy**, then click **Submit** to complete the application. You will be asked to confirm your email address.

Essentials Level Accounts and the Twitter v1.1 APIs

As of mid-2022, Twitter requires new developer accounts to use the Twitter v2 APIs. However, Twitter has not yet migrated some v1.1 APIs to v2. For this reason, Section 12.11's trending-topics examples use the v1.1 APIs. **Essentials-level accounts cannot use the Twitter v1.1 APIs**, but you can apply for an Elevated account to get access to them. If you already had a Twitter developer account before Twitter implemented the v2 API requirement, your account is automatically at the Elevated level.

12.4 Getting Twitter Credentials—Creating an App

Once you have a Twitter developer account, you must obtain **credentials** for interacting with the Twitter APIs. To do so, you'll create a **project** and an **app** within that project. Each app has separate credentials. To create an app, log into

```
https://developer.twitter.com/portal/dashboard
```

and perform the following steps:

1. In your dashboard, click **+ Create Project**
2. Enter a **Project name**. We entered `DeitelTest`. Click **Next**.
3. Select a **Use case**. We selected **Exploring the API**. Click **Next**.
4. In the **Project description** step, describe what you intend to do with your project. We entered "`Experimenting with the Twitter v2 APIs using the examples in the textbook Intro to Python for Computer Science and Data Science: Learning to Program with AI, Big Data and the Cloud`." Click **Next**.
5. In the **Add an existing App or create a new App** step, click **Create new**.
6. In the **App Environment** step, select **Development** (this is the default). Click **Next**.
7. In the **App name** step, specify an app name. We entered `DeitelTestApp`. Click **Next**. Keep this page open for the moment.

Getting Your Credentials

After completing *Step 7* above, you'll see a page titled **Here are your keys & tokens** showing your **Consumer API keys**—the **API Key** and **API Key Secret**—and a **Bearer Token**. Either the API keys or the bearer token can be used to authenticate with Twitter. According to

```
https://developer.twitter.com/en/docs/authentication/oauth-2-0/
   bearer-tokens
```

bearer tokens are more secure, so we'll use the bearer token in this chapter. Click **Copy** to the right of the bearer token's lengthy alphanumeric string.

Storing Your Credentials

As a good practice, do not include your API keys or bearer token (or any other credentials, like usernames and passwords) directly in your source code, as that would expose them to anyone reading the code. You should store your keys in a separate file and never share that file with anyone.[4]

The code you'll execute in subsequent sections assumes that you place your bearer token in the file `keys.py` shown below. You can find this file in the `ch13` examples folder:

```
bearer_token='YourBearerToken'
mapquest_key='YourAPIKey'
```

Open the `keys.py` file in a text editor, select `YourBearerToken` inside the `bearer_token` string and paste your unique bearer token inside the quotes. Ensure you do not have any extra spaces before or after the bearer token inside the string's quotes. Then, save the file and keep it open, as you'll add another API key momentarily.

OAuth 2.0

The API keys or bearer token can be used in the **OAuth 2.0** authentication process[5,6]—known as the "OAuth dance"—that Twitter requires to access its APIs. With Tweepy, you'll provide a bearer token, and it will handle the authentication details for you.

12.5 What's in a Twitter API Response?

The Twitter API methods return **JSON (JavaScript Object Notation)** objects. JSON is a human-readable and computer-readable, text-based data-interchange format used to represent objects as collections of name–value pairs. JSON is commonly used when invoking web services to send and receive across the Internet.

JSON objects are similar to Python dictionaries. Each JSON object contains a list of property-name strings and corresponding values in the following curly braced format:

{*propertyName1*: *value1*, *propertyName2*: *value2*}

As in Python, JSON lists are comma-separated values in square brackets:

[*value1*, *value2*, *value3*]

For your convenience, Tweepy handles the JSON for you behind the scenes, converting JSON to Python objects using classes defined in the Tweepy library.

Default Properties of a Tweet Object

When you acquire a tweet, Twitter returns a JSON object that, by default, contains the tweet's unique ID number and its text (up to a maximum of 280 characters).

4. Good practice would be to use an encryption library to encrypt your keys, bearer tokens and other credentials, then read them in and decrypt them only as you pass them to Twitter.
5. `https://developer.twitter.com/en/docs/authentication/overview`. Accessed August 25, 2022.
6. `https://oauth.net/`. Accessed August 25, 2022.

Twitter Metadata and the Twitter v1.1 APIs

In the Twitter v1.1 APIs, a tweet's JSON object automatically included many additional **metadata** attributes that described aspects of the tweet, such as:

- when it was created,
- who created it,
- lists of the hashtags, URLs, @-mentions and media (such as images and videos) included in the tweet,
- and more.

A typical tweet's JSON object typically contained up to 9,000 characters of metadata—also called the **payload**—which was often far more than your app needed.

Twitter v2 API Expansions and Fields

When you call a Twitter v2 API method, you use **fields** and **expansions**[7] to request the precise metadata your app requires. **Fields** are additional metadata attributes you'd like Twitter to return to your app. For example, when you get a tweet, you might need

- the unique `author_id` attribute, indicating a tweet's sender, or
- the tweet's `created_at` attribute, indicating when the user sent the tweet was sent.

For the complete list of tweet fields, visit

```
https://developer.twitter.com/en/docs/twitter-api/data-dictionary/
   object-model/tweet
```

Some fields are associated with other Twitter metadata objects that, in turn, have their own fields. For example, associated with a tweet's unique `author_id` attribute is a user JSON object. You use an **Expansion** to request that Twitter include associated metadata objects you'd like Twitter to return to your app. Each associated object will contain its default attributes—for a user object, these would be the user's unique `id` number, `name` and `username`, but you can request more. The complete list of user fields can be viewed at

```
https://developer.twitter.com/en/docs/twitter-api/data-dictionary/
   object-model/user
```

For a general overview of all the JSON objects that Twitter APIs return, and links to the specific object details, see

```
https://developer.twitter.com/en/docs/twitter-api/data-dictionary/
   introduction
```

Sample JSON for the NASA Account's 10 Most Recent Tweets

Here's a portion of the JSON from a Twitter API response to a request that asked for recent tweets from the `@NASA` Twitter account. We added line numbers, reformatted the JSON for readability and show two tweets returned. The online Twitter API documentation for each method explains its response.[8]

7. `https://developer.twitter.com/en/docs/twitter-api/data-dictionary/using-fields-and-expansions`. Accessed August 25, 2022.
8. Data obtained on August 24, 2022.

```
{
  "data": [
    {
      "id": "1562156100136292352",
      "text": "RT @NASAInSight: Thanks again for all the kind thoughts
                you've been sending. There's still time to write me a note
                for the mission team to…"
    },
    {
      "id": "1561886047331487744",
      "text": "We see Martian dust devils (whirlwinds) from the ground, as
                in this shot from the Opportunity rover in 2016, left. From
                space, we can see the tracks they leave behind, as in this
                view of dunes from Mars Reconnaissance Orbiter in 2009,
                right. More: https://t.co/kd1BNEDBUD https://t.co/
                RxeKTI5Fv5"
    },
    ...
  ],
  "meta": {
    "result_count": 10,
    "newest_id": "1562156100136292352",
    "oldest_id": "1555635141728382976",
    "next_token": "7140dibdnow9c7btw422nm76p6owdso7rqahg96mulyd2"
  }
}
```

12.6 Installing Tweepy, geopy, folium and deep-translator

We'll use the Tweepy library[9]—one of the most popular Python libraries for interacting with the Twitter APIs.[10] Tweepy makes it easy to access Twitter's capabilities and hides from you the complexities of processing the JSON objects returned by the Twitter APIs. You can view Tweepy's documentation at

```
https://docs.tweepy.org/en/stable/
```

Installing Tweepy

To install Tweepy, open your Anaconda Prompt (Windows), Terminal (macOS/Linux) or shell (Linux), then execute the following command:

```
pip install tweepy
```

Windows users might need to run the Anaconda Prompt as an Administrator. To do so, right-click Anaconda Prompt in the start menu and select **More > Run as administrator.**

Installing geopy

As you work with Tweepy, you'll also use functions from our `tweetutilities.py` file (provided with this chapter's example code). One of the utility functions used by the

9. `https://www.tweepy.org/`. Accessed August 25, 2022.
10. For many additional libraries, see `https://developer.twitter.com/en/docs/twitter-api/tools-and-libraries/v2#python`. Accessed August 25, 2022.

example in Section 12.15 depends on the **geopy library** (`https://github.com/geopy/geopy`) to translate locations into latitude and longitude coordinates—known as **geocoding**—so we can place markers on a map. The library supports dozens of geocoding web services, many of which have free or lite tiers. In Section 12.15, we'll use the **OpenMapQuest geocoding service** (discussed next). To install geopy, execute:

```
conda install -c conda-forge geopy
```

OpenMapQuest Geocoding API

In Section 12.15, we'll use the OpenMapQuest Geocoding API to convert locations, such as Boston, MA, into their latitudes and longitudes, such as 42.3602534 and -71.0582912, for plotting on maps. OpenMapQuest currently allows 15,000 transactions per month on their free tier. To use the service, first sign up at

```
https://developer.mapquest.com/
```

Once logged in, go to

```
https://developer.mapquest.com/user/me/apps
```

and click **Create a New Key**, fill in the **App Name** field with a name of your choosing, leave the **Callback URL** empty and click **Create App** to create an API key. Next, click your app's name to see your consumer key. In the `keys.py` file, store the consumer key by replacing *YourKeyHere* in the line

```
mapquest_key = 'YourKeyHere'
```

You'll import `keys.py` to access this key.

Folium Library and Leaflet.js JavaScript Mapping Library

For the maps in Section 12.15, we'll use the **`folium` library**

```
https://github.com/python-visualization/folium
```

which uses the popular Leaflet.js JavaScript mapping library to display maps. The maps `folium` produces are saved as HTML files that you can view in your web browser. To install `folium`, execute the following command:

```
pip install folium
```

Maps from OpenStreetMap.org

By default, Leaflet.js uses open-source maps from `OpenStreetMap.org`. These maps are copyrighted by the OpenStreetMap.org contributors. To use these maps[11], they require the following copyright notice:

```
Map data © OpenStreetMap contributors
```

and they state:

> *You must make it clear that the data is available under the Open Database License. This can be achieved by providing a "License" or "Terms" link which links to `www.openstreetmap.org/copyright` or `www.opendatacommons.org/licenses/odbl`.*

11. `https://wiki.osmfoundation.org/wiki/Licence/Licence_and_Legal_FAQ`. Accessed August 25, 2022.

deep-translator Library

People tweet in many languages. We'll use the **deep-translator library**[12]—which supports several translation services—to translate foreign-language tweets into English via Google Translate. To install `deep-translator`, use:

```
pip install -U deep_translator
```

12.7 Authenticating with Twitter Via Tweepy to Access Twitter v2 APIs

In the next several sections, you'll invoke various cloud-based Twitter APIs via Tweepy. Here you'll use Tweepy to authenticate with Twitter and create a **Tweepy `Client` object**, your gateway to using the Twitter v2 APIs over the Internet. In subsequent sections, you'll work with various Twitter APIs by invoking methods on your `Client` object.

Before you invoke any Twitter API, you must use your bearer token to authenticate with Twitter.[13] Launch IPython from the `ch13` examples folder, then import **`tweepy`** and the `keys.py` file you modified earlier in this chapter. You can import any `.py` file as a module by using the file's name *without* the `.py` extension in an `import` statement:

```
In [1]: import tweepy

In [2]: import keys
```

When you import `keys.py` as a module, you can individually access each variable defined in that file as `keys.`*variable_name*.

Creating a Client Object

To use the Twitter v2 APIs, you must first create a Tweepy `Client` object, initializing it with your bearer token:

```
In [3]: client = tweepy.Client(bearer_token=keys.bearer_token,
   ...:                        wait_on_rate_limit=True)
```

We specified two arguments in this call to the `Client` constructor:

- `bearer_token` is the bearer token you acquired in Section 12.4 to authenticate with Twitter.
- `wait_on_rate_limit=True` tells Tweepy that each time it reaches a given API method's rate limit it should wait for the rate-limit interval to expire. This ensures that you do not violate Twitter's rate-limit restrictions. For most Twitter APIs, the rate-limit interval is 15 minutes.

You're now ready to interact with Twitter via Tweepy. The code examples in the next several sections are presented as a continuous IPython session, so the authorization process you went through here need not be repeated.

12. `https://deep-translator.readthedocs.io/en/latest/`. Accessed August 25, 2022.
13. For apps that enable users to log into their Twitter accounts, manage them, post tweets, read tweets from other users, search for tweets, etc., you'll need user authentication rather than app authentication. For details on user authentication with Tweepy, see `https://docs.tweepy.org/en/latest/authentication.html`. Accessed August 25, 2022.

12.8 Getting Information About a Twitter Account

After authenticating with Twitter, you can use the Tweepy `Client` object's **`get_user` method** to get a **`tweepy.Response` object** containing information about a user's Twitter account. Let's get information about NASA's `@NASA` Twitter account:

```
In [4]: nasa = client.get_user(username='NASA',
   ...:     user_fields=['description', 'public_metrics'])
```

The `get_user` method with the `username` keyword argument calls the Twitter API's

```
/2/users/by/username/:username
```

method,[14] which returns JSON data that Tweepy converts into a `tweepy.Response` object. We'll say more about this object momentarily.

Twitter returns the account's ID number, name and user name by default. Twitter API methods that return user account information enable you to request additional user account fields. In Tweepy, you specify these fields via the `user_fields` keyword argument. Here we requested the account's `description` and `public_metrics`, which we'll discuss momentarily. The complete list of user fields can be viewed at:

```
https://developer.twitter.com/en/docs/twitter-api/data-dictionary/
   object-model/user
```

Each Twitter method has a rate limit. For example, you can call Twitter's

```
/2/users/by/username/:username
```

method up to 900 times every 15 minutes to get information on specific user accounts. As we mention other methods, we'll provide a footnote with a link to each method's documentation in which you can view its limits.

`tweepy.Response` Object

Each `tweepy.Response` object contains four fields:

- `data`—the data returned by Twitter.
- `includes`—additional data specified via a given method's expansions parameter.
- `errors`—information about the errors that occurred, if any.
- `meta`—method-specific information that can be useful in processing the response.

Getting a User's Basic Account Information

Let's display some information about the `@NASA` account. When a Twitter method returns a user JSON object, the Tweepy `Response` object's `data` attribute is a named tuple containing the default fields `id`, `name` and `username`:

- The **`id`** is the account's unique ID number.
- The **`name`** is the name associated with the user's account.
- The **`username`** is the user's Twitter handle (`@NASA`). For NASA, both have the same value, but `name` often represents a user's actual name. To protect a user's privacy, the `name` and `username` values are sometimes created names.

14. `https://developer.twitter.com/en/docs/twitter-api/users/lookup/api-reference/get-users-by-username-username`. Accessed August 25, 2022.

We also requested the additional `user_fields` `description` and `public_metrics`, so these, too, are in the `Response` object's `data` attribute. The **`description`** contains the text description provided in the user's profile. We discuss the `public_metrics` below.

```
In [5]: nasa.data.id
Out[5]: 11348282

In [6]: nasa.data.name
Out[6]: 'NASA'

In [7]: nasa.data.username
Out[7]: 'NASA'

In [8]: nasa.data.description
Out[8]: "There's space for everybody."
```

Getting the Number of Accounts That Follow This Account and the Number of Accounts This Account Follows

A user account's **`public_metrics`** attribute is a dictionary containing the keys:

- `'followers_count'`—the number of users who follow this account,
- `'following_count'`—the number of users that this account follows,
- `'tweet_count'`—the total number of tweets (and retweets) sent by this user, and
- `'listed_count'`—the total number of Twitter lists that include this user.

Here we show just the `'followers_count'` and `'following_count'`:

```
In [9]: nasa.data.public_metrics['followers_count']
Out[9]: 61260251

In [10]: nasa.data.public_metrics['following_count']
Out[10]: 181
```

Getting Your Own Account's Information

You can also use the properties in this section on your account. To do so, call the Tweepy `Client` object's **`get_me` method**, as in:

```
me = client.get_me()
```

This returns a `User` object for the account you used to authenticate with Twitter in the preceding section. As with get_users, you may specify arguments to request additional information about the account.

12.9 Intro to Tweepy Paginators: Getting More than One Page of Results

When invoking Twitter APIs, you often receive as results collections of objects, such as tweets sent by a particular user, tweets matching specified search criteria or tweets in a user's **timeline** (consisting of tweets sent by a user and by other accounts that user follows).

Each Twitter API method's documentation discusses the maximum number of items the method can return per call—this is known as a **page** of results. When you request more results than a given method can return, Twitter's JSON response contains information to help you manage requests for the additional pages. Tweepy's `Paginator` handles these details for you. A **Paginator**[15] invokes a specified `Client` method and checks whether there is another page of results. If so, the `Paginator` automatically calls the method again to get those results. This continues (subject to the method's rate limits) until there are no more results to process. If you configure the `Client` object to wait when rate limits are reached (as we did), the `Paginator` will adhere to the rate limits and wait as needed between calls. The following subsections discuss `Paginator` fundamentals.

12.9.1 Determining an Account's Followers

Let's use a Tweepy `Paginator` to invoke the `Client` object's **`get_users_followers` method**, which calls the Twitter API's

```
/2/users/:id/followers
```

method[16] to obtain an account's followers. Twitter returns these in groups of 100 by default, but you can request up to 1000 at a time. For demonstration purposes, we'll grab 10 of NASA's followers, five at a time, so we receive two pages of results. Let's begin by creating a list in which we'll store the followers' Twitter user names:

```
In [17]: followers = []
```

Creating a Paginator

Next, let's create a `Paginator` object that will call the `get_users_followers` method for NASA's account:

```
In [18]: paginator = tweepy.Paginator(
    ...:     client.get_users_followers, nasa.data.id, max_results=5)
```

You initialize the `Paginator` with the name of the method to call and any arguments that should be passed to that method:

- `client.get_users_followers` indicates that the `Paginator` will call the `client` object's `get_users_followers` method,
- `nasa.data.id` is the ID number (obtained in Section 12.8) of the NASA Twitter account for which we'll get followers, and
- `max_results=5` specifies that each page of results should contain five followers.

Getting Results

Now, we can use the `Paginator` to get some followers. The following `for` statement iterates through the results of the expression `paginator.flatten(10)`. The `Paginator`'s **`flatten` method** initiates the call to `client.get_users_followers`. The argument 10 indicates the total number of results to obtain. We iterate through these and add each follower's `username` to the `followers` list:

```
In [19]: for follower in paginator.flatten(limit=10):
    ...:     followers.append(follower.username)
    ...:
```

15. `https://docs.tweepy.org/en/latest/v2_pagination.html`. Accessed August 25, 2022.
16. `https://developer.twitter.com/en/docs/twitter-api/users/follows/api-reference/get-users-id-followers`. Accessed August 25, 2022.

Let's display the followers in ascending order:

```
In [20]: print('Followers:',
    ...:       ' '.join(sorted(followers, key=lambda s: s.lower())))
Followers: ARNOLDO81766323 CusumanoNolan desthiafh egrh50686195 epic90for
F1lukesuperfan GreenTolland misra_arsh RpKumbhar98 virendrarathv17
```

We call the built-in `sorted` function with the second argument specifying how the elements of `followers` are sorted. In this case, the `lambda` converts every user name to lowercase letters so we can perform a case-insensitive sort.

Automatic Paging

If the number of results requested is more than one call to `get_users_followers` returns, the `flatten` method automatically "pages" through the results by making multiple calls to `client.get_users_followers`. We specified in snippet `[18]` that each page contains five results, so snippet `[19]` will get two pages of results. Method `flatten` makes the two pages appear to be a sequence of 10 results.

If you do not specify an argument to the `flatten` method, the `Paginator` attempts to get all of the account's followers. This could take significant time due to Twitter's rate limits. The Twitter method[17] called by `get_users_followers` can return a maximum of 1000 followers at a time, and Twitter allows up to 15 calls every 15 minutes. Thus, you can only get 15,000 followers every 15 minutes using Twitter's free APIs. Recall that we configured the `Client` object to automatically wait when it hits a rate limit. So if you try to get all followers and an account has more than 15,000, Tweepy will automatically pause for 15 minutes after every 15,000 followers and display a message. You saw in snippet `[9]` that, at the time of this writing, NASA had over 61 million followers. At 60,000 followers per hour, it would take over 40 days to get all of NASA's followers.

Note that for this example, we could have simply called `get_users_followers` since we're getting only a small number of followers. We used a `Paginator` here to show how you'll typically call `Client` methods. In subsequent examples, we'll call `Client` methods directly to get just a few results, rather than using `Paginator`s.

12.9.2 Determining Whom an Account Follows

The `Client` object's **`get_users_following` method** calls the Twitter API's

```
/2/users/:id/following
```

method[18] to get a list of Twitter users an account follows. Twitter returns these in groups of 100 by default, but you can request up to 1000 at a time. You can call this method up to 15 times every 15 minutes. Let's get 10 accounts that NASA follows:

```
In [25]: following = []

In [26]: paginator = tweepy.Paginator(
    ...:     client.get_users_following, nasa.data.id, max_results=5)
```

17. `https://developer.twitter.com/en/docs/twitter-api/users/follows/api-reference/get-users-id-followers`. Accessed August 25, 2022.
18. `https://developer.twitter.com/en/docs/twitter-api/users/follows/api-reference/get-users-id-following`. Accessed August 25, 2022.

```
In [27]: for user_followed in paginator.flatten(limit=10):
    ...:     following.append(user_followed.username)
    ...:

In [28]: print('Following:',
    ...:       ' '.join(sorted(following, key=lambda s: s.lower())))
Following: Astro_Ayers astro_berrios astro_deniz astro_matthias Astro_Pam
astro_watkins JimFree NASA_Gateway NASASpaceSci v_wyche
```

12.9.3 Getting a User's Recent Tweets

The `Client` method **`get_users_tweets`** returns a `tweepy.Response` containing tweets from a specified user. The method calls the Twitter API's

`/2/users/:id/tweets`

method[19], which returns the most recent 10 tweets but can between 5 and 100 at a time. This method can return only an account's 3200 most recent tweets. Applications using this method may call it up to 1500 times every 15 minutes.

The `data` attribute of the `tweepy.Response` returned by `get_users_tweets` contains a list of the returned tweets. Each object in that list has a `data` attribute, which is a dictionary containing the keys `'id'` and `'text'` for each tweet's unique ID and its text. Let's display five tweets from the `@NASA` account using its ID number that we obtained previously:

```
In [29]: nasa_tweets = client.get_users_tweets(
    ...:     id=nasa.data.id, max_results=5)

In [30]: for tweet in nasa_tweets.data:
    ...:     print(f"NASA: {tweet.data['text']}\n")
    ...:
NASA: Come find out how college students are getting involved in
developing and testing technologies for future Moon missions.

Join the livestream on @Twitch today at 4pm ET (2000 UTC) and chat with
teams from this year's @NASAArtemis Student Challenges: https://t.co/
6EOhJoy2TD https://t.co/OF1RFnu6qD

NASA: #Artemis I is "go" for launch! Now that today's flight readiness
review has concluded, NASA managers provide an update on the Moon
mission, scheduled to lift off at 8:33am ET (12:33 UTC), Monday, Aug. 29.
More info: https://t.co/KOrOCmSRu4 https://t.co/apV6wrEYCu

NASA: RT @NASAArtemis: Update: Today's flight readiness review briefing
on the #Artemis I mission is now scheduled for 8pm ET (00:00 UTC).
Watch:…

NASA: @enrosadire @NASAArtemis The many moods of @NASAMoon. ?

NASA: @profdanthomas It's always fun to draw the Moon! Thank you for
sharing, Oscar!
```

In snippet `[29]`, we called the `get_users_tweets` method directly and used the keyword argument `max_results` to specify the number of tweets to retrieve. If you wish to get

19. `https://developer.twitter.com/en/docs/twitter-api/tweets/timelines/api-reference/get-users-id-tweets`. Accessed August 25, 2022.

more than the maximum number of tweets per call (`100`), then you should use a `Paginator` to call `get_users_tweets`, as shown in Section 12.9.

Grabbing Recent Tweets from Your Own Timeline

You can call the `Client` method **`get_home_timeline`**, as in:

```
client.get_home_timeline()
```

to get tweets from your home timeline[20]—that is, your tweets and retweets, as well as tweets and retweets from the Twitter users you follow. This method calls Twitter's

```
/2/users/:id/timelines/reverse_chronological
```

method[21] and returns up to a maximum of 100 tweets by default. For more than that, you should use a Tweepy `Paginator` to call `get_home_timeline`.

12.10 Searching Recent Tweets; Intro to Twitter v2 API Search Operators

The Tweepy `Client` method **`search_recent_tweets`** returns tweets from the last seven days that match a query string you provide. The method calls Twitter's

```
/2/tweets/search/recent
```

method[22], which returns a minimum of 10 tweets at a time (the default) but can return up to 100 (specified with keyword argument `max_results`). Use a `Paginator` if you need more results than can be returned by one `search_recent_tweets` call. It's possible that fewer than 10 tweets will match the specified query string.

Utility Function `print_tweets` from `tweetutilities.py`

For this section, we created a utility function `print_tweets` (in `tweetutilities.py`) that receives the results of a call to `Client` method `search_recent_tweets` and displays for each tweet the tweeter's `username` and the tweet's `text`. If the tweet is not in English and the `tweet.lang` is not `'und'` (undefined), we also translate the tweet to English using the `deep-translator` library's `GoogleTranslator` class, which `tweetutilities.py` imports.[23] The `GoogleTranslator` object's translate function receives ISO 639-1 language codes[24] for a `source` and a `target` language—`source='auto'` enables Google to auto-detect the source language. To use `print_tweets`, import it from `tweetutilities.py`:

```
In [33]: from tweetutilities import print_tweets
```

Just the `print_tweets` function's definition from that file is shown below—we'll explain the `tweets` parameter's contents (used in line 8) momentarily:

20. Specifically for the account you used to authenticate with Twitter.
21. `https://developer.twitter.com/en/docs/twitter-api/tweets/timelines/api-reference/get-users-id-reverse-chronological`. Accessed August 25, 2022.
22. `https://developer.twitter.com/en/docs/twitter-api/tweets/search/api-reference/get-tweets-search-recent`. Accessed August, 25, 2022.
23. `https://github.com/nidhaloff/deep-translator`. Accessed August, 25, 2022.
24. https://en.wikipedia.org/wiki/List_of_ISO_639-1_codes. Accessed August, 27, 2022.

```
 1  def print_tweets(tweets):
 2      # translator to autodetect source language and return English
 3      translator = GoogleTranslator(source='auto', target='en')
 4
 5      """For each tweet in tweets, display the username of the sender
 6      and tweet text. If the language is not English, translate the text
 7      with the deep-translator library's GoogleTranslator."""
 8      for tweet, user in zip(tweets.data, tweets.includes['users']):
 9          print(f'{user.username}:', end=' ')
10
11          if 'en' in tweet.lang:
12              print(f'{tweet.text}\n')
13          elif 'und' not in tweet.lang: # translate to English first
14              print(f'\n  ORIGINAL: {tweet.text}')
15              print(f'TRANSLATED: {translator.translate(tweet.text)}\n')
```

Searching for Specific Words

Let's call the `Client` object's `search_recent_tweets` method to search for 10 recent tweets about the Webb Space Telescope. The method returns a `Response` object in which the `data` attribute contains a list of matching tweets:

```
In [34]: tweets = client.search_recent_tweets(
    ...:     query='Webb Space Telescope',
    ...:     expansions=['author_id'], tweet_fields=['lang'])
```

The `query` keyword argument specifies the query string containing your search criteria. Twitter returns only each tweet's unique ID and text by default. In this example, we'd like to show who sent the tweet and check the tweet's language so we can decide whether to translate it. The language (`'lang'`) is an additional field you may request via the list you provide in the `tweet_fields` parameter. You can view the complete list of tweet fields at:

```
https://developer.twitter.com/en/docs/twitter-api/data-dictionary/
   object-model/tweet
```

As we mentioned in Section 12.5, the Twitter v2 API also supports **expansions**, which enable you to request related metadata objects to be included in a method's response. The expansion `'author_id'` indicates that for each tweet, Twitter also should return the user JSON object for the user who sent the tweet. As discussed in Section 12.8, user JSON object contains the user's `id`, `name` and `username` by default. If you need more user fields, you can pass a list to the `user_fields` parameter shown previously. Tweepy places the expansion objects in the `Response`'s `includes` dictionary attribute. For the `'author_id'` expansion, a list of tweet authors is stored with the key `'users'`. Each tweet has a corresponding user in this list. So the following expression in line 8 of `print_tweets`:

```
zip(tweets.data, tweets.includes['users'])
```

creates tuples in which the first element represents a tweet (from the list `tweets.data`) and the second element represents the user object for the sender (from the list stored in the `tweets.includes` dictionary's `'users'` key). Snippet `[35]` displays the tweets—we showed just two of the results to save space:

```
In [35]: print_tweets(tweets)
John11110111101: RT @uhd2020: Zoom Into the Southern Ring Nebula Captured
by NASA James Webb Space Telescope https://t.co/CWR8LOwN5d
```

```
zeeejayee: RT @SpaceTelescope: After years of preparation and
anticipation, exoplanet researchers are ecstatic! The James Webb Space
Telescope has cap…
```

Note that one of these tweets was a retweet, as indicated by `RT` at the beginning of the tweet. We'll show how to check whether a tweet is a retweet and ignore it later.

Searching with Twitter v2 API Search Operators

You can use various Twitter search operators[25] in your query strings to refine your search results. Your query-string length is limited by your developer account type:

- For Essentials and Elevated accounts, query strings may be up to 512 characters.
- For Academic Research accounts, query strings may be up to 1024 characters.

Also, some operators are available only for Elevated accounts or higher.

The Twitter v2 operators are categorized as **standalone** or **conjunction-required**:

- **Standalone operators** can be used alone or combined with other operators in a query string.
- **Conjunction-required** operators must be combined with at least one standalone operator in a query string. Otherwise, Twitter says conjunction-required operators would match "an extremely high volume of Tweets."

The following table shows several Twitter search operators, as well as logical AND, logical OR and logical negation capabilities. As with Python code, parentheses can be used to group query-string subexpressions. All matching is performed using case-insensitive searching, so searching for `Python` can also return matches for `python`.

Example	Finds tweets containing
`python twitter`	Finds tweets containing `python` AND `twitter`. Spaces between query string terms and operators are implicitly treated as logical AND operations. In this query string, `python` and `twitter` are terms to search for—these are considered **standalone operators**.
`python OR twitter`	Finds tweets containing `python` OR `twitter` OR both. **The logical `OR` operator is case-sensitive.**
`planets -mars`	`-` (minus sign)—Finds tweets containing `planets` but not `mars`. The minus is the logical NOT operator and can be applied to any operator.
An emoji	You can use emojis as standalone operators in a query string to find tweets containing those emojis.
`has:hashtags`, `has:links`, `has:mentions`, `has:media`, …	You can combine these conjunction-required operators with standalone operators to find tweets containing hashtags, links, mentions of other users, media and more.

25. `https://developer.twitter.com/en/docs/twitter-api/tweets/search/integrate/build-a-query`. Accessed August 25, 2022.

Example (Cont.)	Finds tweets containing
`is:retweet`, `is:reply`, `is:verified`, …	You can combine these conjunction-required operators with standalone operators to determine whether a tweet is a retweet, a tweet is a reply, the sender is a verified Twitter account and more.
`place:"New York City"`	Finds tweets that were sent near `"New York City"`. Multiword places should be quoted as shown here.
`from:NASA`	Finds tweets from the account `@NASA`.
`to:NASA`	Finds tweets to the account `@NASA`. You also may use `to:`*`id`*, where *`id`* is the unique ID number of the user account.

Operator Documentation and Tutorial

You can view all the operators with examples of each at

```
https://developer.twitter.com/en/docs/twitter-api/tweets/search/
   integrate/build-a-query
```

Check out Twitter's tutorial on building high-quality Twitter v2 API query strings to obtain the targeted results your app requires:

```
https://developer.twitter.com/en/docs/tutorials/building-high-
   quality-filters
```

Twitter also provides an online tool to help you build Twitter v2 API query strings:

```
https://developer.twitter.com/apitools/query?query=
```

Searching for Tweets From NASA Containing Links

Let's use the `from` and `has:links` operators to get recent tweets from NASA that contain hyperlinks:

```
In [36]: tweets = client.search_recent_tweets(
    ...:     query='from:NASA has:links',
    ...:     expansions=['author_id'], tweet_fields=['lang'])

In [37]: print_tweets(tweets)
NASA: Come find out how college students are getting involved in
developing and testing technologies for future Moon missions.

Join the livestream on @Twitch today at 4pm ET (2000 UTC) and chat with
teams from this year's @NASAArtemis Student Challenges: https://t.co/
6EOhJoy2TD https://t.co/OF1RFnu6qD
```

Searching for a Hashtag

Tweets often contain **hashtags** that begin with # to indicate something of importance, like a trending topic. Let's get tweets containing the hashtag `#metaverse`—we showed just two results to save space:

```
In [38]: tweets = client.search_recent_tweets(query='#metaverse',
    ...:     expansions=['author_id'], tweet_fields=['lang'])
    ...:

In [39]: print_tweets(tweets)
```

```
adamrbses: @shush_club @Safelaunch1 @BabylonsNFT BINECD CORP MEGA
Hello everyone, binecd will like to inform and engage the general public
on it ongoing investment plan and giveaway projects, kindly follow the
link and make sure to participate.
https://t.co/eNmxhZt9WZ
#CryptoGiveaway #Metaverse #BTC #Giveaway

CsmicCouncil: Wen Metaverse??

As a new development with a wealth of unrealized potential, the hype
around the #Metaverse is expected.

However, we will focus (for now) on phygital methods that will forge
external and cultural connections, creating a community where Cosmics can
thrive.
```

12.11 Spotting Trending Topics

[Note: At the time of this writing, Twitter had not yet migrated their Trending Topics APIs from v1.1 to v2. The v1.1 APIs used in this section are accessible only to Twitter Developer accounts with "Elevated" access and higher.]

If a topic "goes viral," thousands or even millions of people could tweet about it. Twitter calls these **trending topics** and maintains lists of them worldwide. Via the Twitter v1.1 Trends API, you can get lists of locations with trending topics and lists of the top 50 trending topics for each location. To use the v1.1 APIs in Tweepy, initialize an object of class **`OAuth2BearerHandler`** with your bearer token, then create an **`API`** object that uses the `OAuth2BearerHandler` object to authenticate with Twitter:

```
In [42]: auth = tweepy.OAuth2BearerHandler(keys.bearer_token)

In [43]: api = tweepy.API(auth=auth, wait_on_rate_limit=True)
```

12.11.1 Places with Trending Topics

The Tweepy `API`'s **`available_trends` method** calls the Twitter v1.1 API's `trends/available`[26] method to get a list of all locations for which Twitter has trending topics. Method `available_trends` returns a list of dictionaries representing these locations. When we executed this code, there were 467 locations with trending topics:

```
In [44]: available_trends = api.available_trends()

In [45]: len(available_trends)
Out[45]: 467
```

The dictionary in each list element returned by `available_trends` has various information, including the location's `name` and `woeid` (discussed below):

```
In [46]: available_trends[0]
Out[46]:
{'name': 'Worldwide',
 'placeType': {'code': 19, 'name': 'Supername'},
 'url': 'http://where.yahooapis.com/v1/place/1',
 'parentid': 0,
```

26. `https://developer.twitter.com/en/docs/twitter-api/v1/trends/locations-with-trending-topics/api-reference/get-trends-available`. Accessed August, 25, 2022.

```
     'country': '',
     'woeid': 1,
     'countryCode': None}

    In [47]: available_trends[1]
    Out[47]:
    {'name': 'Winnipeg',
     'placeType': {'code': 7, 'name': 'Town'},
     'url': 'http://where.yahooapis.com/v1/place/2972',
     'parentid': 23424775,
     'country': 'Canada',
     'woeid': 2972,
     'countryCode': 'CA'}
```

The Twitter v1.1 API's `trends/place` method (discussed momentarily) uses **Yahoo! Where on Earth IDs (WOEIDs)** to look up trending topics. The WOEID `1` represents worldwide, and other locations have unique WOEID values greater than `1`. We'll use WOEID values in the following two subsections to get worldwide trending topics and trending topics for a specific city. The following table shows WOEID values for several landmarks, cities, states and continents. Although these are valid WOEIDs, Twitter does not necessarily have trending topics for all these locations.

Place	WOEID	Place	WOEID
Statue of Liberty	23617050	Iguazu Falls	468785
Los Angeles, CA	2442047	United States	23424977
Washington, D.C.	2514815	North America	24865672
Paris, France	615702	Europe	24865675

You also can search for locations close to a location that you specify with latitude and longitude values. To do so, call the Tweepy `API`'s **`closest_trends` method**, which invokes the Twitter API's `trends/closest` method.[27]

12.11.2 Getting a List of Trending Topics

The Tweepy `API`'s **`get_place_trends` method** calls the Twitter Trends API's `trends/place` method[28] to get the top 50 trending topics for the specified WOEID. You can get WOEIDs from the `woeid` attribute in each dictionary returned by the `available_trends` or `closest_trends` methods discussed in the previous section, or you can find a location's WOEID by searching for a city/town, state, country, address, zip code or landmark at

    http://www.woeidlookup.com

You also can look up WOEID's programmatically using Yahoo!'s web services via Python libraries like `woeid`:

    https://github.com/Ray-SunR/woeid

27. `https://developer.twitter.com/en/docs/twitter-api/v1/trends/locations-with-trending-topics/api-reference/get-trends-closest`. Accessed August, 25, 2022.
28. `https://developer.twitter.com/en/docs/twitter-api/v1/trends/trends-for-location/api-reference/get-trends-place`.Accessed August, 25, 2022.

Worldwide Trending Topics

Let's get today's worldwide trending topics (your results will differ):

```
In [48]: world_trends = api.get_place_trends(id=1)
```

Method `get_place_trends` returns a one-element list containing a dictionary in which the `'trends'` key refers to a list of dictionaries representing each trend:

```
In [49]: trends_list = world_trends[0]['trends']
```

Each trend dictionary has `name`, `url`, `promoted_content` (indicating the tweet is an advertisement), `query` and `tweet_volume` keys (shown below). The following trend is a hashtag:

```
In [50]: trends_list[0]
Out[50]:
{'name': '#SOUMUN',
 'url': 'http://twitter.com/search?q=%23SOUMUN',
 'promoted_content': None,
 'query': '%23SOUMUN',
 'tweet_volume': 121659}
```

You'll often see a mix of hashtags and phrases in many languages in the trending topics.

For trends with more than 10,000 tweets, the `tweet_volume` is the number of tweets; otherwise, it's `None`. Let's use a list comprehension to filter the list so that it contains only trends with more than 10,000 tweets:

```
In [51]: trends_list = [t for t in trends_list if t['tweet_volume']]
```

Next, let's sort the trends in descending order by `tweet_volume`:

```
In [52]: from operator import itemgetter

In [53]: trends_list.sort(key=itemgetter('tweet_volume'), reverse=True)
```

Now, let's display the names of the top five trending topics:

```
In [54]: for trend in trends_list[:5]:
    ...:     print(trend['name'])
    ...:
DONBELLE PHIHNOMENALConcert
Southampton
#SOUMUN
KANAWUT
#LetsGULFtoJAPAN
```

New York City Trending Topics

Now, let's get the top five trending topics for New York City (WOEID `2459115`). The following code performs the same tasks as above, but for the different WOEID:

```
In [55]: nyc_trends = api.get_place_trends(id=2459115)

In [56]: nyc_list = nyc_trends[0]['trends']

In [57]: nyc_list = [t for t in nyc_list if t['tweet_volume']]

In [58]: nyc_list.sort(key=itemgetter('tweet_volume'), reverse=True)

In [59]: for trend in nyc_list[:5]:
    ...:     print(trend['name'])
    ...:
```

```
    #MUFC
    Chelsea
    Ronaldo
    Nigeria
    Southampton
```

12.11.3 Create a Word Cloud from Trending Topics

In the NLP chapter, we used the WordCloud library to create word clouds. Let's use it here to visualize New York City's trending topics with more than 10,000 tweets each. First, let's create a dictionary of key–value pairs consisting of the trending topic `names` and `tweet_volumes`:

```
In [65]: topics = {}

In [66]: for trend in nyc_list:
    ...:     topics[trend['name']] = trend['tweet_volume']
    ...:
```

Next, let's create a `WordCloud` from the `topics` dictionary's key–value pairs, then output the word cloud to the image file `TrendingTwitter.png` (shown after the code). The argument `prefer_horizontal=0.5` suggests that 50% of the words should be horizontal, though the software may ignore that to fit the content:

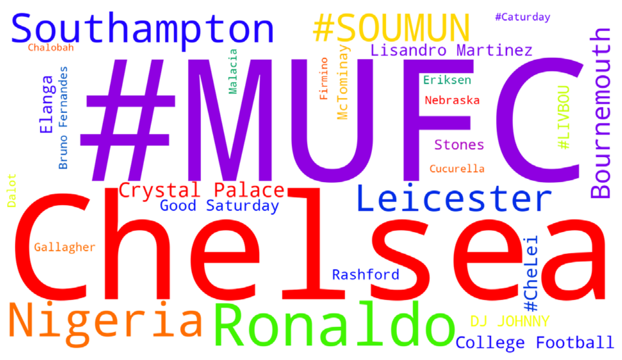
```
In [67]: from wordcloud import WordCloud

In [68]: wordcloud = WordCloud(width=1600, height=900,
    ...:     prefer_horizontal=0.5, min_font_size=10, colormap='prism',
    ...:     background_color='white')
    ...:

In [69]: wordcloud = wordcloud.fit_words(topics)

In [70]: wordcloud = wordcloud.to_file('TrendingTwitter.png')
```

The resulting word cloud is shown below—yours will differ based on the trending topics the day you run the code:



12.12 Cleaning/Preprocessing Tweets for Analysis

Data cleaning is one of the most common tasks that data scientists perform. Depending on how you intend to process tweets, you'll need to use natural language processing to normalize them by performing various data cleaning tasks in the following table. Many of these can be performed using the libraries introduced in the "Natural Language Processing (NLP)" chapter:

Tweet cleaning tasks	
Converting all text to the same case	Removing stop words
Removing the # symbol from hashtags	Removing RT (retweet) and FAV (favorite)
Removing @-mentions	Removing URLs
Removing duplicates	Stemming
Removing excess whitespace	Lemmatization
Removing hashtags	Tokenization
Removing punctuation	

tweet-preprocessor Library and TextBlob Utility Functions

In this section, we'll use the **tweet-preprocessor library**

```
https://github.com/s/preprocessor
```

to perform some basic tweet cleaning. It can automatically remove any combination of:

- URLs,
- @-mentions (like `@nasa`),
- hashtags (like `#mars`),
- Twitter reserved words (like `RT` for retweet and `FAV` for favorite, which is similar to a "like" on other social networks),
- emojis (all or just smileys) and
- numbers

The following table shows the module's constants representing each option:

Option	Option constant	Option	Option constant
@-Mentions (e.g., `@nasa`)	`OPT.MENTION`	Reserved Words (`RT` and `FAV`)	`OPT.RESERVED`
Emoji	`OPT.EMOJI`	Smiley	`OPT.SMILEY`
Hashtag (e.g., `#mars`)	`OPT.HASHTAG`	URL	`OPT.URL`
Number	`OPT.NUMBER`		

Installing tweet-preprocessor

To install tweet-preprocessor, open your Anaconda Prompt (Windows), Terminal (macOS/Linux) or shell (Linux), then issue the following command:

```
pip install tweet-preprocessor
```

Windows users might need to run the Anaconda Prompt as an administrator for proper software installation privileges. To do so, right-click Anaconda Prompt in the start menu and select **More > Run as administrator**.

Cleaning a Tweet

Let's do some basic tweet cleaning that we'll use in a later example in this chapter. The tweet-preprocessor library's module name is `preprocessor`. Its documentation recommends that you import the module as follows:

```
In [1]: import preprocessor as p
```

To set the cleaning options you'd like to use, call the module's **`set_options`** function. In this case, we'd like to remove URLs and Twitter reserved words:

```
In [2]: p.set_options(p.OPT.URL, p.OPT.RESERVED)
```

Now let's clean a sample tweet containing a reserved word (RT) and a URL:

```
In [3]: tweet_text = 'RT A sample retweet with a URL https://nasa.gov'

In [4]: p.clean(tweet_text)
Out[4]: 'A sample retweet with a URL'
```

12.13 Twitter Streaming API

Your app can receive tweets as they occur in real-time. Based on the Twitter Statistics page at `InternetLiveStats.com`,[29] we calculated that there are over 10,000 tweets per second and approximately 880 million tweets per day.[30] Most developer accounts are subject to a **tweet cap**[31]—a maximum number of tweets per month that an account's Twitter apps can acquire using the Twitter APIs. The tweet caps are 500,000 for Essentials accounts and two million for Elevated accounts—academic research and paid accounts can get more.

This section uses a class definition and an IPython session to process streaming tweets. Note that the code for receiving a tweet stream requires creating a custom class that inherits from another class. These topics are covered in Chapter 10.

12.13.1 Creating a Subclass of `StreamingClient`

The Streaming API returns tweets as they happen. Rather than connecting to Twitter on each method call, a stream uses a persistent connection to **push** (that is, send) tweets to your app. The rate at which those tweets arrive varies tremendously based on your search criteria, which you'll specify with Tweepy **`StreamRule`** objects. The more popular a topic is, the more likely tweets will arrive quickly. Twitter uses all the `StreamRule`s you set to find tweets, including `StreamRule`s you've set previously. So you may want to delete existing `StreamRule`s before creating new ones, as you'll see in Section 12.13.2.

You create a subclass of Tweepy's **`StreamingClient` class** to process the tweet stream. Tweepy calls the methods on an object of this class as it receives each new tweet (or other message, such as an error) from Twitter. For example,

- `on_connect(self)` is called when the app successfully connects to the Twitter stream.
- `on_respone(self, response)` is called when a response arrives from the Twitter stream—the `response` parameter is a Tweepy **`StreamResponse`** named tuple object containing the tweet data, any expansion objects you requested and more.

29. `http://www.internetlivestats.com/twitter-statistics/`. Accessed August 25, 2022.
30. As of August 2022.
31. `https://developer.twitter.com/en/docs/twitter-api/tweet-caps`. Accessed August 25, 2022.

StreamingClient already defines these and other "on_" methods. You override (redefine) only the methods your app needs. For additional StreamingClient methods, see:

    https://docs.tweepy.org/en/latest/streamingclient.html

Class TweetListener

Our StreamingClient subclass TweetListener is defined in tweetlistener.py. Line 6 indicates that class TweetListener is a subclass of tweepy.StreamingClient. This ensures that our new class has class StreamingClient's default method implementations.

```
1  # tweetlistener.py
2  """StreamingClient subclass that processes tweets as they arrive."""
3  from deep_translator import GoogleTranslator
4  import tweepy
5
6  class TweetListener(tweepy.StreamingClient):
7      """Handles incoming Tweet stream."""
8
```

Class TweetListener: __init__ Method

Class TweetListener's __init__ method is called when you create a new TweetListener object. The bearer_token parameter is used to authenticate with Twitter. The limit parameter is the number of tweets to process—10 by default. We added this parameter so you can control the number of tweets to receive. As you'll see, we terminate the stream when that limit is reached. Line 11 creates an instance variable to track the number of tweets processed so far, and line 12 creates a constant to store the limit. Line 15 creates a GoogleTranslator object for translating tweets into English. If you're not familiar with __init__ and super() from previous chapters, line 17 passes the bearer_token to the superclass's __init__, which authenticates with Twitter. **We also set wait_on_rate_limit=True to ensure that we do not violate the Twitter rate limits for our account type.**

```
9      def __init__(self, bearer_token, limit=10):
10         """Create instance variables for tracking number of tweets."""
11         self.tweet_count = 0
12         self.TWEET_LIMIT = limit
13
14         # GoogleTranslator object for translating tweets to English
15         self.translator = GoogleTranslator(source='auto', target='en')
16
17         super().__init__(bearer_token, wait_on_rate_limit=True)
18
```

Class TweetListener: on_connect Method

Method **on_connect** is called when your app successfully connects to the Twitter stream. We override the default implementation to display a "Connection successful" message.

```
19     def on_connect(self):
20         """Called when your connection attempt is successful, enabling
21         you to perform appropriate application tasks at that point."""
22         print('Connection successful\n')
23
```

Class TweetListener: on_response Method

Method **`on_response`** is called by Tweepy when each tweet arrives. This method's second parameter is a Tweepy **`StreamResponse`** named tuple object containing:

- `data`—the tweet's attributes.
- `includes`—any requested expansion objects.
- `errors`—any errors that occurred.
- `matching_rules`—the specific `StreamRules` that the returned tweet matched.

As you'll see, we'll use an expansion (Section 12.5) to include in the `StreamResponse` the **user JSON object** for each tweet's sender. Interestingly, Twitter also returns user objects for accounts mentioned in the tweet's text. Line 29 gets the sender's `username`, which is stored in list element `0` of `response.includes['users']`. Subsequent elements would contain accounts mentioned in the tweet. Lines 30–32 display the tweet sender's `username` and the tweet's language (`lang`) and `text`. If the language is not English (`'en'`) and not undefined (`'und'`), lines 34–36 translate the tweet to English and display it. Line 39 increments `self.tweet_count`. Lines 45–46 determine whether to terminate streaming.

```
24      def on_response(self, response):
25          """Called when Twitter pushes a new tweet to you."""
26  
27          try:
28              # get username of user who sent the tweet
29              username = response.includes['users'][0].username
30              print(f'Screen name: {username}')
31              print(f'   Language: {response.data.lang}')
32              print(f' Tweet text: {response.data.text}')
33  
34              if response.data.lang != 'en' and response.data.lang != 'und':
35                  english = self.translator.translate(response.data.text)
36                  print(f' Translated: {english}')
37  
38              print()
39              self.tweet_count += 1
40          except Exception as e:
41              print(f'Exception occured: {e}')
42              self.disconnect()
43  
44          # if TWEET_LIMIT is reached, terminate streaming
45          if self.tweet_count == self.TWEET_LIMIT:
46              self.disconnect()
```

12.13.2 Initiating Stream Processing

Let's use an IPython session to obtain tweets using a `TweetListener` object. First, import Tweepy and the `keys.py` file:

```
In [1]: import tweepy

In [2]: import keys
```

Creating a TweetListener

The `StreamingClient` subclass `TweetListener` manages the connection to the Twitter stream and receives and processes the tweets. Create a `TweetListener` object, initializing it with your bearer token and the number of tweets you'd like to receive (3) before the TweetListener terminates the connection:

```
In [3]: from tweetlistener import TweetListener

In [4]: tweet_listener = TweetListener(
   ...:     bearer_token=keys.bearer_token, limit=3)
```

Redirecting the Standard Error Stream to the Standard Output Stream

When you eventually call your StreamingClient subclass's disconnect method to terminate the tweet stream, the method sends the message

```
Stream connection closed by Twitter
```

to the standard error stream (sys.stderr), which is not synchronized with the standard output stream. Sometimes, this causes the preceding message to be interspersed with other messages that this app sends to the standard output stream. To prevent this, redirect the standard error stream to the standard output stream:

```
In [5]: import sys

In [6]: sys.stderr = sys.stdout
```

Deleting Existing Stream Rules

When you initiate the tweet stream, Twitter uses all the StreamRules you've specified previously to filter the tweets it pushes to your app—that is, it sends you only tweets that match the search criteria specified in the StreamRules. Twitter does not automatically remove your StreamRules after you terminate the tweet stream. If your app filters the tweet stream with different rules each time you run it, you should delete any existing StreamRules before creating new ones. To do so:

1. Get the StreamRules by calling your StreamingClient's **get_rules** method—the Response's data attribute contains a list of StreamRules:

```
In [7]: rules = tweet_listener.get_rules().data
```

2. Get the rule IDs—here, we use a list comprehension to create a list containing all the existing rules' IDs:

```
In [8]: rule_ids = [rule.id for rule in rules]
```

3. Call your StreamingClient's **delete_rules** method, which receives a list of rule IDs to delete. This method's response contains a 'summary' dictionary with information about the number of deleted rules.

```
In [9]: tweet_listener.delete_rules(rule_ids)
Out[9]: Response(data=None, includes={}, errors=[], meta={'sent': '2022-
08-23T23:50:51.138Z', 'summary': {'deleted': 1, 'not_deleted': 0}})
```

Creating and Adding a Stream Rule

In this example, we'd like to filter the live tweet stream, looking for tweets about football. To do so, create a StreamRule:

```
In [10]: filter_rule = tweepy.StreamRule('football')
```

Next, call your StreamingClient's **add_rules** method, passing the StreamRule (or a list of StreamRules as an argument:

```
In [11]: tweet_listener.add_rules(filter_rule)
Out[11]: Response(data=[StreamRule(value='football', tag=None,
id='1562225901483483137')], includes={}, errors=[], meta={'sent': '2022-
08-23T23:50:55.945Z', 'summary': {'created': 1, 'not_created': 0,
'valid': 1, 'invalid': 0}})
```

This method's `Response` contains a `'summary'` dictionary with information about the `StreamRule` you just set and whether it was valid.

Starting the Tweet Stream

The `Stream` object's **`filter` method** begins the streaming process. Here, we use the keyword argument `expansions` to indicate that we'd like the response for each tweet to include the sender's user JSON object. The keyword argument `tweet_fields` indicates that the tweet's language should be included in the responses tweet JSON object:

```
In [12]: tweet_listener.filter(
    ...:     expansions=['author_id'], tweet_fields=['lang'])
```

The following output shows three streamed tweets:

```
Connection successful

Screen name: MikeRebello1
   Language: en
 Tweet text: Pilgrim Football live from camp fogarty https://t.co/
VTOX6RMJ3F

Screen name: ChazJ
   Language: en
 Tweet text: @blue_gwladys Pro Football players are assets the same as
the floodlights and the chairman's office chair. Everything has a price.
I think Spurs did us over re Richy but for 60M Chelsea US are madder than
Chelski were. Grab it. Two strikers and a #10 window closed.

Screen name: JamesCDolan92
   Language: en
 Tweet text: @EduardoHagn 9/10 be 2 massive signings to a already great
attack arteta building a really good team there and are playing some good
football things are looking up for arsenal fans so far

Stream connection closed by Twitter
```

Asynchronous vs. Synchronous Streams

Tweepy supports **asynchronous tweet streams** by creating a subclass of **`AsyncStreamingClient` class**. This allows your application to continue executing while your listener waits to receive tweets. Asynchronous streams are convenient in GUI applications, so users can continue interacting with other parts of the application while tweets arrive.

12.14 Tweet Sentiment Analysis

In the NLP chapter, we demonstrated sentiment analysis on sentences. Many researchers and companies perform sentiment analysis on tweets. For example, political researchers might check tweet sentiment during election seasons to understand how people feel about specific politicians and issues. Companies might check tweet sentiment to see what people say about their products and competitors' products.

Let's use the techniques introduced in the preceding section to create a script (`sentimentlistener.py`) that checks the sentiment on a specific topic. The script will keep totals of all the positive, neutral and negative tweets it processes and display the results.

The script receives two command-line arguments representing the topic of the tweets you wish to receive and the number of tweets for which to check the sentiment. Only those tweets that are not eliminated are counted. For viral topics, there are large numbers of retweets, which we are not counting, so it could take some time to get the number of tweets you specify. You can run the script from the `ch13` folder as follows:

```
ipython sentimentlistener.py football 10
```

which produces output like the following. Positive tweets are preceded by a `+`, negative tweets by a `-` and neutral tweets by a space:

```
  smfalk: 'What a difference a year makes' for Red Bank Regional football
program via @asburyparkpress

- MarieInSedona: @MollyJongFast His base is trapped in a USFL Fantasy
Football league. They are bored, disappointed and ready to trade.

  _ethannn: @Chace_THFC @DavidTa41816701 @paarsons @RobertAllen97 did
spurs create football chants?

+ wassimfcb23: Football is much more than a game

  zaimmzaidi: @90min_Football: Adama Traore is back!

- PhiloeEsq: 1 Euopa final, 3 UCL finals, lost 2 to Madrid. + 2
ridiculous 2nd place finishes in the league. That’s without putting into
context the style of football he implemented. Let’s behave like adults,
please.

+ x_hems: @BYUDFO: When I was 16 years old I wrote down my life goals…
One of them being to be on staff of a Top 25 NCAA Division I Football
Team.…

+ wocoblanco: @FootballMissess: Football fans are the best

+ NovieRohani: @Hector_Network: We’re Champion Partner of
#BorussiaDortmund! #BVB is one of the most iconic football clubs in the
world! Follow us for…

+ tsloan_17: It's about that time. On the call tomorrow for
@ProsperEaglesFB vs @IAR2_Football on @sportsgram. Kickoff at 7:00 from
Pennington Field. Pregame Show at 6:45. High School Football is back, and
this is as fun a matchup as you can draw up to open the season!

Stream connection closed by Twitter
Tweet sentiment for "football"
Positive: 5
 Neutral: 3
Negative: 2
```

Sentiment analysis is not a perfect process. Do you agree with these sentiment characterizations? The script (`sentimentlistener.py`) is presented below. We focus only on the new capabilities in this example.

Imports

Lines 4–8 import the `keys.py` file and the libraries used throughout the script:

```
1  # sentimentlisener.py
2  """Script that searches for tweets that match a search string
3  and tallies the number of positive, neutral and negative tweets."""
4  import keys
5  import preprocessor as p
6  import sys
7  from textblob import TextBlob
8  import tweepy
9
```

Class `SentimentListener`: `__init__` Method

In addition to the `bearer_token` for authenticating with Twitter, the `__init__` method receives three additional parameters:

- `sentiment_dict`—a dictionary in which we'll keep track of the tweet sentiments,
- `topic`—the topic we're searching for so we can ensure it's in the tweet text and
- `limit`—the number of tweets to process (not including the ones we eliminate).

Each of these is stored in the current `SentimentListener` object (`self`).

```
10 class SentimentListener(tweepy.StreamingClient):
11     """Handles incoming Tweet stream."""
12
13     def __init__(self, bearer_token, sentiment_dict, topic, limit=10):
14         """Configure the SentimentListener."""
15         self.sentiment_dict = sentiment_dict
16         self.tweet_count = 0
17         self.topic = topic
18         self.TWEET_LIMIT = limit
19
20         # set tweet-preprocessor to remove URLs/reserved words
21         p.set_options(p.OPT.URL, p.OPT.RESERVED)
22         super().__init__(bearer_token, wait_on_rate_limit=True)
23
```

Method `on_response`

If the tweet is not a retweet (line 28), line 29 gets and cleans the tweet to remove URLs and Twitter reserved words. Lines 32–33 skip the tweet if the topic is not in the text. Lines 36–45 use a `TextBlob` to check the tweet's sentiment and update the `sentiment_dict` accordingly.Line 48 gets the sender's `username` from `response.includes['users']`—as you'll see when we start the streaming, we'll use an expansion to include this user object. Line 49 prints the tweet text preceded by + for positive sentiment, space for neutral sentiment or - for negative sentiment. Line 51 increments the `tweet_count`, and lines 54–55 check whether the app should disconnect from the tweet stream.

```
24     def on_response(self, response):
25         """Called when Twitter pushes a new tweet to you."""
26
27         # if the tweet is not a retweet
28         if not response.data.text.startswith('RT'):
29             text = p.clean(response.data.text) # clean the tweet
```

```
30
31             # ignore tweet if the topic is not in the tweet text
32             if self.topic.lower() not in text.lower():
33                 return
34
35             # update self.sentiment_dict with the polarity
36             blob = TextBlob(text)
37             if blob.sentiment.polarity > 0:
38                 sentiment = '+'
39                 self.sentiment_dict['positive'] += 1
40             elif blob.sentiment.polarity == 0:
41                 sentiment = ' '
42                 self.sentiment_dict['neutral'] += 1
43             else:
44                 sentiment = '-'
45                 self.sentiment_dict['negative'] += 1
46
47             # display the tweet
48             username = response.includes['users'][0].username
49             print(f'{sentiment} {username}: {text}\n')
50
51             self.tweet_count += 1 # track number of tweets processed
52
53             # if TWEET_LIMIT is reached, terminate streaming
54             if self.tweet_count == self.TWEET_LIMIT:
55                 self.disconnect()
56
```

Main Application

The main application is defined in the function `main` (lines 57–87; discussed after the following code), which is called by lines 90–91 when you execute the file as a script. So `sentimentlistener.py` can be imported into IPython or other modules to use class `SentimentListener` as we did with `TweetListener` in the previous section:

```
57 def main():
58     # get search term and number of tweets
59     search_key = sys.argv[1]
60     limit = int(sys.argv[2]) # number of tweets to tally
61
62     # set up the sentiment dictionary
63     sentiment_dict = {'positive': 0, 'neutral': 0, 'negative': 0}
64
65     # create the StreamingClient subclass object
66     sentiment_listener = SentimentListener(keys.bearer_token,
67         sentiment_dict, search_key, limit)
68
69     # redirect sys.stderr to sys.stdout
70     sys.stderr = sys.stdout
71
72     # delete existing stream rules
73     rules = sentiment_listener.get_rules().data
74     rule_ids = [rule.id for rule in rules]
75     sentiment_listener.delete_rules(rule_ids)
76
```

```
77      # create stream rule
78      sentiment_listener.add_rules(
79          tweepy.StreamRule(f'{search_key} lang:en'))
80
81      # start filtering English tweets containing search_key
82      sentiment_listener.filter(expansions=['author_id'])
83
84      print(f'Tweet sentiment for "{search_key}"')
85      print('Positive:', sentiment_dict['positive'])
86      print(' Neutral:', sentiment_dict['neutral'])
87      print('Negative:', sentiment_dict['negative'])
88
89  # call main if this file is executed as a script
90  if __name__ == '__main__':
91      main()
```

In `main`, lines 59–60 get the command-line arguments. Line 63 creates the `sentiment_dict` dictionary that keeps track of the tweet sentiments. Lines 66–67 create the `SentimentListener`. Line 70 redirects the standard error stream to the standard output stream. Lines 73–75 delete any existing `StreamRules`. Lines 78–79 create a new `StreamRule` that searches for English (`lang:en`) tweets that match the `search_key`. Line 82 starts the stream. The `expansions` parameter indicates that we'd like Twitter to include the tweet sender's user object in the response. Once the tweets have been received and processed, lines 84–87 display the sentiment report.

12.15 Geocoding and Mapping

In this section, we'll collect streaming tweets, then plot the locations of those tweets. Most tweets do not include latitude and longitude coordinates because Twitter disables this by default for all users. Those who wish to include their precise location in tweets must enable that feature. A large percentage of tweets include the user's home location information. However, even that is sometimes invalid, such as "Far Away" or a fictitious location from a user's favorite movie.

In this section, for simplicity, we'll use the location stored in the Twitter account that sent each tweet to plot that user's location on an interactive map. The map will let you zoom in and out and drag to move the map around so you can look at different areas (known as **panning**). For each tweet, we'll display a map marker that you can click to see a pop-up containing the user's screen name and tweet text.

We'll ignore retweets and tweets that do not contain the search topic. For other tweets, we'll track the percentage for which the sender's account contains location information. When we get the latitude and longitude information for those locations, we'll also track the percentage of those tweets with invalid location data.

12.15.1 Getting and Mapping the Tweets

Let's interactively develop the code that plots tweet locations. We'll use utility functions from our `tweetutilities.py` file and class `LocationListener` in `locationlistener.py`. We'll explain the utility functions and `LocationListener` details.

Collections Required By `LocationListener`

Our `LocationListener` class requires two collections—a list (`tweets`) to store the data from the tweets we collect, and a dictionary (`counts`) to track the total number of tweets we collect and the number that have location data:

```
In [1]: tweets = []

In [2]: counts = {'total_tweets': 0, 'locations': 0}
```

Creating the `LocationListener`

For this example, the `LocationListener` will collect 50 tweets about `'football'`:

```
In [3]: import keys

In [4]: import tweepy

In [5]: from locationlistener import LocationListener

In [6]: location_listener = LocationListener(
   ...:     keys.bearer_token, counts_dict=counts, tweets_list=tweets,
   ...:     topic='football', limit=50)
```

The `LocationListener` will use our utility function `get_tweet_content` (located in `tweetutilities.py`; discussed in Section 12.15.2) to place in a dictionary the username, tweet text and user location from each tweet.

Redirect `sys.stderr` to `sys.stdout`

As in the previous two examples, we redirect the standard error stream to the standard output stream so the message `"Stream connection closed by Twitter"` that displays when we disconnect from the tweet stream does not get interspersed with other text sent to the standard output stream:

```
In [7]: import sys

In [8]: sys.stderr = sys.stdout
```

Delete Existing `StreamRules`

Once again, Twitter applies all the rules that you've set previously unless you delete them:

```
In [9]: rules = location_listener.get_rules().data

In [10]: rule_ids = [rule.id for rule in rules]

In [11]: location_listener.delete_rules(rule_ids)
Response(data=None, includes={}, errors=[], meta={'sent': '2022-08-
22T21:16:18.357Z', 'summary': {'deleted': 1, 'not_deleted': 0}})
```

Create a `StreamRule`

In this example, we'll get tweets in English (`lang:en`) about football:

```
In [12]: location_listener.add_rules(
    ...:     tweepy.StreamRule('football lang:en'))
Response(data=[StreamRule(value='football lang:en', tag=None,
id='1561824608181010432')], includes={}, errors=[], meta={'sent': '2022-
08-22T21:16:19.955Z', 'summary': {'created': 1, 'not_created': 0,
'valid': 1, 'invalid': 0}})
```

Configure and Start the Stream of Tweets

Next, let's start streaming the tweets:

```
In [13]: location_listener.filter(expansions=['author_id'],
    ...:     user_fields=['location'], tweet_fields=['lang'])
```

The expansion `'author_id'` gets information about the user who sent the tweet, including the username. The `user_fields` argument specifies that the user information should include the account's `'location'`. The `tweet_fields` argument specifies additional information to include with each tweet—in this case, the tweet's language.

Now, wait to receive the tweets. Though we do not show them here (to save space), the `LocationListener` displays each tweet's screen name and text so you can see them as they arrive from the live stream. If you're not receiving any (perhaps it is not football season), you might want to type *Ctrl* + *C* to terminate the previous snippet, delete the `StreamRule` and set up a new one for a different topic.

Displaying the Location Statistics

When the next `In []` prompt displays, we can check how many tweets we processed, how many had locations and the percentage that had locations:

```
In [14]: counts['total_tweets']
Out[14]: 83

In [15]: counts['locations']
Out[15]: 50

In [16]: print(f'{counts["locations"] / counts["total_tweets"]:.1%}')
60.2%
```

In this particular execution, 60.2% of the tweets contained location data.

Geocoding the Locations

Now, let's use our `get_geocodes` utility function (from `tweetutilities.py`; discussed in Section 12.15.2) to geocode the location of each tweet stored in the list of `tweets`:

```
In [17]: from tweetutilities import get_geocodes

In [18]: bad_locations = get_geocodes(tweets)
Getting coordinates for tweet locations...
OpenMapQuest service timed out. Waiting.
OpenMapQuest service timed out. Waiting.
Done geocoding
```

Sometimes the OpenMapQuest geocoding service times out, meaning that it cannot handle your request immediately, and you need to try again. In that case, our function `get_geocodes` would display

```
OpenMapQuest service timed out. Waiting.
```

wait for a short time, then retry the geocoding request.

As you'll soon see, for each tweet with a valid location, the `get_geocodes` function adds the new keys `'latitude'` and `'longitude'` to that tweet's dictionary in the `tweets` list. For their values, the function uses the coordinates that OpenMapQuest returns. These will be used to plot map markers on our interactive map.

Displaying the Bad Location Statistics

When the next `In []` prompt displays, we can check the percentage of tweets that had invalid location data:

```
In [19]: bad_locations
Out[19]: 9

In [20]: print(f'{bad_locations / counts["locations"]:.1%}')
18.0%
```

In this case, 9 of the 50 (18%) tweets we acquired for which the sender's account contained a location had invalid locations.

Cleaning the Data

Before we plot the tweet locations on a map, let's use a pandas `DataFrame` to clean the data. When you create a `DataFrame` from the `tweets` list, it will contain the value `NaN` for the `'latitude'` and `'longitude'` of any tweet that does not have a valid location. Since `NaN` cannot be plotted on a map, let's remove any rows containing `NaN` by calling the `DataFrame`'s **`dropna` method**:

```
In [21]: import pandas as pd

In [22]: df = pd.DataFrame(tweets)

In [23]: df = df.dropna()
```

Creating a Map with Folium

Next, let's create a folium **`Map`** on which we'll plot the tweet locations:

```
In [24]: import folium

In [25]: usmap = folium.Map(location=[39.8283, -98.5795],
    ...:     tiles='Stamen Terrain', zoom_start=5, detect_retina=True)
```

The `location` keyword argument specifies a sequence containing latitude and longitude coordinates for the map's center point. The values in this snippet are the geographic center of the continental United States.[32] In many places worldwide, the term `'football'` describes the sport we call soccer in the U.S., so some of the tweets we plot may be outside the U.S. In this case, you will not see them initially when you open the map. You can zoom using the `+` and `-` buttons at the map's top-left, or you can dragging the map with the mouse (that is, pan) to see anywhere in the world.

The `zoom_start` keyword argument specifies the map's initial zoom level, lower values show more of the world, and higher values show less. On our system, `5` displays the entire continental United States. The `detect_retina` keyword argument enables folium to detect high-resolution screens. When it does, it requests higher-resolution maps from OpenStreetMap.org and changes the zoom level accordingly.

Creating Popup Markers for the Tweet Locations

Next, we'll create folium `Popup` objects containing each tweet's text and add them to the `Map`. To do so, let's iterate through the `DataFrame` one row at a time. `DataFrame` method **`itertuples`** creates a named tuple from each row. Each named tuple will contain properties corresponding to each `DataFrame` column:

32. `https://bit.ly/CenterOfTheUS`.

```
In [26]: for t in df.itertuples():
    ...:     text = ': '.join([t.username, t.text])
    ...:     popup = folium.Popup(text, parse_html=True)
    ...:     marker = folium.Marker((t.latitude, t.longitude),
    ...:                            popup=popup)
    ...:     marker.add_to(usmap)
    ...:
```

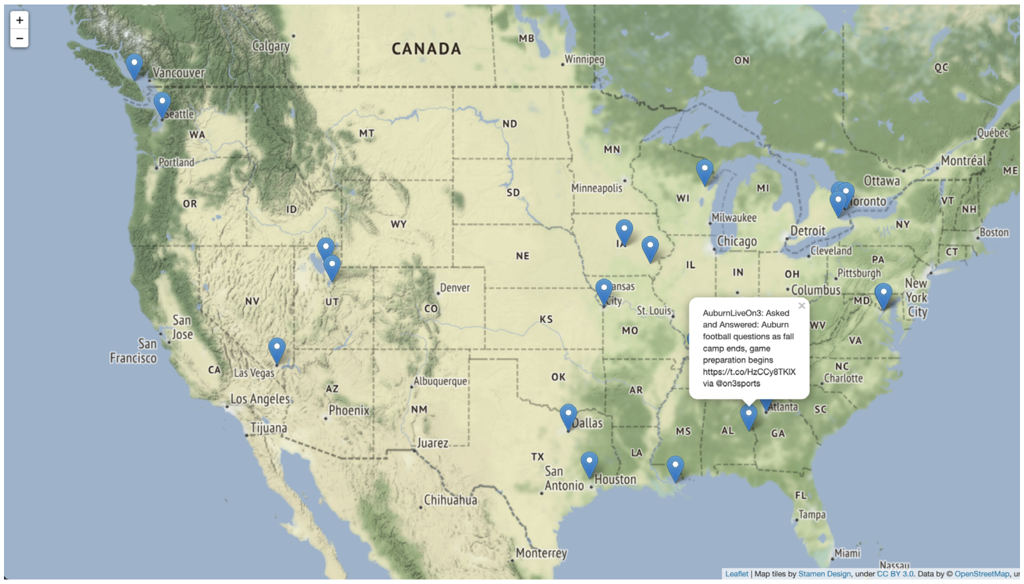
First, we create a string (`text`) containing the user's `username` and tweet `text` separated by a colon and a space. This text will be displayed on the map in a popup if you click the corresponding marker. The second statement creates a folium **`Popup`** to display the text. The third statement creates a folium **`Marker`**, using a tuple to specify the `Marker`'s latitude and longitude. The `popup` keyword argument associates the tweet's `Popup` object with the new `Marker`. Finally, the last statement calls the `Marker`'s **`add_to` method** to specify the `Map` that will display the `Marker`.

Saving the Map

The last step is to call the `Map`'s **`save`** method to store the map in an HTML file, which you can then double-click to open in your web browser:

```
In [27]: usmap.save('tweet_map.html')
```

The resulting map follows. The `Marker` positions on your map will differ:



AuburnLiveOn3: Asked and Answered: Auburn football questions as fall camp ends, game preparation begins https://t.co/HzCCy8TKIX via @on3sports

Map data © OpenStreetMap contributors.
The data is available under the Open Database License `www.openstreetmap.org/copyright`.

12.15.2 Utility Functions in `tweetutilities.py`

Here we present the utility functions `get_tweet_content` and `get_geocodes` used in the preceding section's IPython session. In each case, the line numbers start from 1 for discussion purposes. These are both defined in `tweetutilities.py`, which is included in the `ch13` examples folder.

get_tweet_content Utility Function

Function `get_tweet_content` receives a `StreamResponse` object containing a tweet's data and other fields we requested via the `StreamingClient` `filter` method's keyword arguments `expansions`, `user_fields` and `tweet_fields`. The function returns a dictionary containing the tweet's `username` (line 4), `text` (line 5) and `location` (line 6):

```
1  def get_tweet_content(response):
2      """Return dictionary with data from tweet."""
3      fields = {}
4      fields['username'] = response.includes['users'][0].username
5      fields['text'] = response.data.text
6      fields['location'] = response.includes['users'][0].location
7
8      return fields
```

get_geocodes Utility Function

Function `get_geocodes` receives a list of dictionaries containing tweets and attempts to geocode their user locations. If geocoding is successful for a given tweet's user location, the function adds the latitude and longitude to the corresponding tweet's dictionary in `tweet_list`. This code requires class **`OpenMapQuest`** from the `geopy` module, which `tweetutilities.py` imports as follows:

```
from geopy import OpenMapQuest
```

```
def get_geocodes(tweet_list):
    """Get the latitude and longitude for each tweet's location.
    Returns the number of tweets with invalid location data."""
    print('Getting coordinates for tweet locations...')
    geo = OpenMapQuest(api_key=keys.mapquest_key)  # geocoder
    bad_locations = 0

    for tweet in tweet_list:
        processed = False
        delay = .1  # used if OpenMapQuest times out to delay next call
        while not processed:
            try:  # get coordinates for tweet['location']
                geo_location = geo.geocode(tweet['location'])
                processed = True
            except:  # timed out, so wait before trying again
                print('OpenMapQuest service timed out. Waiting.')
                time.sleep(delay)
                delay += .1

        if geo_location:
            tweet['latitude'] = geo_location.latitude
            tweet['longitude'] = geo_location.longitude
        else:
            bad_locations += 1  # tweet['location'] was invalid

    print('Done geocoding')
    return bad_locations
```

The function operates as follows:

- Line 5 creates the `OpenMapQuest` object we'll use to geocode locations. The `api_key` keyword argument is loaded from the `keys.py` file you edited in Section 12.6.
- Line 6 initializes `bad_locations`, which we use to keep track of the number of invalid locations in the tweet objects we collected.
- In the loop, lines 9–18 attempt to geocode the current tweet's location. As we mentioned, the OpenMapQuest geocoding service will sometimes time out, meaning it's temporarily unavailable. This can happen if you make too many requests too quickly. For this reason, the `while` loop continues executing as long as `processed` is `False`. Each iteration of this loop calls the `OpenMapQuest` object's **geocode method** with the tweet's user location as an argument. If successful, processed is set to `True`, and the loop terminates. Otherwise, lines 16–18 display a time-out message, tell the loop to wait for `delay` seconds and increase the delay in case of another time-out. Line 17 calls the Python Standard Library `time` module's `sleep` method to pause the code execution.
- After the `while` loop terminates, lines 20–24 check whether location data was returned and, if so, add it to the tweet's dictionary. Otherwise, line 24 increments the `bad_locations` counter.
- Finally, the function prints a message that it's done geocoding and returns the `bad_locations` value.

12.15.3 Class `LocationListener`

Class `LocationListener` performs many of the same tasks we demonstrated in the previous streaming examples, so we'll focus on just a few lines in this class:

```python
# locationlistener.py
"""Receives tweets matching a search string and stores a list of
dictionaries containing each tweet's username/text/location."""
import tweepy
from tweetutilities import get_tweet_content

class LocationListener(tweepy.StreamingClient):
    """Handles incoming Tweet stream to get location data."""

    def __init__(self, bearer_token, counts_dict,
                 tweets_list, topic, limit=10):
        """Configure the LocationListener."""
        self.tweets_list = tweets_list
        self.counts_dict = counts_dict
        self.topic = topic
        self.TWEET_LIMIT = limit
        super().__init__(bearer_token, wait_on_rate_limit=True)

    def on_response(self, response):
        """Called when Twitter pushes a new tweet to you."""

        # get each tweet's username, text and location
        tweet_data = get_tweet_content(response)
```

```
24
25          # ignore retweets and tweets that do not contain the topic
26          if (tweet_data['text'].startswith('RT') or
27              self.topic.lower() not in tweet_data['text'].lower()):
28              return
29
30          self.counts_dict['total_tweets'] += 1 # it's an original tweet
31
32          # ignore tweets with no location
33          if not tweet_data.get('location'):
34              return
35
36          self.counts_dict['locations'] += 1 # user account has location
37          self.tweets_list.append(tweet_data) # store the tweet
38          print(f"{tweet_data['username']}: {tweet_data['text']}\n")
39
40          # if TWEET_LIMIT is reached, terminate streaming
41          if self.counts_dict['locations'] == self.TWEET_LIMIT:
42              self.disconnect()
```

Again, the `__init__` method receives the `bearer_token` and the number of tweets to process (`limit`). In this example, `__init__` also receives a `counts` dictionary that we use to keep track of the total number of tweets processed, a `tweet_list` in which we store the dictionaries returned by the `get_tweet_content` utility function, and a string representing the `topic` so we can confirm that its text is contained in the tweet text.

In method `on_response`:

- Line 23 calls `get_tweet_content` to get each tweet's screen name, text and location.
- Lines 26–28 ignore the tweet if it is a retweet or if the text does not include the topic we're searching for. We'll use only original tweets containing the search string.
- Line 30 adds 1 to the value of the `'total_tweets'` key in the `counts` dictionary to track the number of original tweets we process.
- Lines 33–334 ignore tweets that have no location data.
- Line 36 adds 1 to the value of the `counts` dictionary's `'locations'` key to indicate that we found a tweet with a location.
- Line 37 appends the `tweet_data` dictionary to the `tweets_list`.
- Line 38 displays the tweet's screen name and tweet text so you can see that the app is making progress.
- Lines 41–42 check whether the `TWEET_LIMIT` has been reached, and if so, disconnect from the stream.

12.16 Storing Tweets

Marketers, researchers and others frequently store tweets they receive from the Streaming API. For analysis, you'll commonly store tweets in:

- CSV files—A file format that we introduced in the "Files and Exceptions" chapter.

- pandas `DataFrames` in memory—CSV files can be loaded easily into `DataFrames` for cleaning and manipulation.
- SQL databases—Such as MySQL, a free and open source relational database management system (RDBMS).
- NoSQL databases—Twitter returns tweets as JSON documents, so the natural way to store them is in a NoSQL JSON document database, such as MongoDB. Tweepy generally hides the JSON from the developer. If you'd like to manipulate the JSON directly, use the techniques we present in the "Big Data: Hadoop, Spark, NoSQL and IoT Databases" chapter, where we'll look at the PyMongo library.

If you store tweets, Twitter requires you to delete any data for which you receive a deletion message. For deletion rules, see

```
https://developer.twitter.com/en/developer-terms/agreement-and-
   policy
```

12.17 Twitter and Time Series

A time series is a sequence of values with timestamps. Some examples are daily closing stock prices, daily high temperatures at a given location, monthly U.S. job-creation numbers, quarterly earnings for a given company and more. Tweets are natural for time-series analysis because they're time stamped. In the "Machine Learning" chapter, we'll use a technique called simple linear regression to make predictions with time series. We'll take another look at time series in the "Deep Learning" chapter when we study recurrent neural networks.

12.18 Wrap-Up

In this chapter, we explored data mining Twitter, perhaps the most open and accessible of all the social media sites, and one of the most commonly used big-data sources. You created a Twitter developer account and connected to Twitter using your account credentials. We discussed Twitter's rate limits and the importance of following their rules.

We showed that the Twitter APIs return responses in JSON format. We used Tweepy—one of the most widely used Twitter API clients—to authenticate with Twitter and access the Twitter v2 APIs. We saw that tweets returned by the Twitter APIs contain default attributes and that we could use the Twitter v2 API's expansions and fields to request additional metadata. We determined an account's followers and whom an account follows, and looked at a user's recent tweets.

We used Tweepy `Paginators` to conveniently request multiple pages of results from various Twitter APIs. We searched for past tweets that met specified criteria. We tapped into the flow of live tweets as they happened with a subclass of Tweepy's `StreamingClient` class. We used the Twitter v1.1 Trends API to determine trending topics for various locations and created a word cloud from trending topics.

We cleaned and preprocessed tweets to prepare them for analysis and performed sentiment analysis on tweets. We used the folium library to create a map of tweet locations and interacted with it to see the tweets at particular locations. We enumerated common ways to store tweets and noted that tweets are a natural form of time series data. The next chapter presents IBM Watson and its cognitive computing capabilities.

Index

Symbols

'author_id' expansion (Twitter v2 API) 366
@-mentions 355

A

Academic Research Twitter developer account level 335, 349
add_rules method of a StreamingClient (Tweepy) **359**
add_to method of class Marker **368**
API **351**
API class (Tweepy) **351**
  available_trends method **351**
  closest_trends method **352**
  trends_place method **352**
API key (Twitter) **336**
API key secret (Twitter) **336**
app for managing Twitter credentials **336**
app rate limit (Twitter API) 334
asynchronous tweet stream **360**
AsyncStreamingClient class (Tweepy) **360**
available_trends method of class API **351**

B

bearer token 341, 351
bearer token (Twitter) **336**

C

case-insensitive sort 345
cleaning data 367
Client class (Tweepy) **341**
  get_me method 343
  get_user method **342**
  get_users_followers method **344**, 344
  get_users_following method **345**
  get_users_tweets method **346**, 346, 347
  home_timeline method **347**
  search method **347**
closest_trends method of class API **352**
cloud 334
conjunction-required Twitter v2 API search operator **349**
constructor 341
Consumer API keys (Twitter) **336**
continental United States 367
coordinates (map) 340
credentials for using the Twitter API **336**

D

data cleaning 354, 367
data mining **332**, 333
  Twitter 332
DataFrame (pandas) 367
  dropna method **367**
  itertuples method **367**
deep-translator library **341**
delete_rules method of a StreamingClient (Tweepy) **359**
description property of a User (Twitter) **343**
disruptive technology **333**
dropna method of class DataFrame **367**

E

Elevated Twitter developer account level 335, 336, 349, 351, 356
endpoint
  of a web service **334**
Essentials Twitter developer account level 335, 349, 356
expansions in a Twitter v2 API method call 338
expansions in the Twitter v2 API 348

F

fields in a Twitter v2 API method call 338
filter method of class Stream **360**
Folium mapping library **340**
  Map class **367**
  Marker class **368**
  Popup class **368**

G

geocode a location 366

geocode method of class OpenMapQuest (geopy) **370**
geocoding **340**, 369
  OpenMapQuest geocoding service **340**
geographic center of the continental United States 367
geopy library **340**
  OpenMapQuest class **369**, **370**
get_me method of class Client 343
get_rules method of a StreamingClient (Tweepy) **359**
get_user method of class Client **342**
get_users_followers method of class Client **344**, 344
get_users_following method of class Client **345**
get_users_tweets method of class Client 346, 347
get_users_tweets method of class Client (Tweepy) **346**

H
hashtags **350**
home timeline 347
home_timeline method of class Client **347**

I
IBM Watson 334
id property of a User (Twitter) **342**
install Tweepy 339, 355
items method
  of Cursor **344**
itertuples method of class DataFrame **367**

J
JavaScript Object Notation (JSON) **337**
JSON (JavaScript Object Notation) **337**
  object 337

L
lambda expression 345
latitude 340
Leaflet.js JavaScript mapping library 340
LinkedIn 333
longitude 340

M
map
  coordinates 340
  marker 364
  panning **364**
  zoom 364
Map class (Folium) **367**
  save method **368**
Marker class (folium) **368**
  add_to method **368**
Meta (formerly called Facebook) 333
metadata
  tweet **338**
microblogging **333**
modules
  tweepy 341
MongoDB document database 372

N
name property of a User (Twitter) **342**
named tuple 367
natural language processing (NLP) 354
NoSQL database 372

O
OAuth 2.0 **337**
OAuth2BearerHandler **351**
OAuth2BearerHandler class (Tweepy) **351**
on_connect method of class StreamingClient 356, **357**
on_response method of class StreamingClient 358
on_status method of class StreamingClient 356
OpenMapQuest
  API key 340
OpenMapQuest (geopy)
  geocode method **370**
OpenMapQuest class (geopy) **369**, **370**
OpenMapQuest geocoding service **340**
OpenStreetMap.org 340
overriding a method 357

P
page of Twitter results 343
Paginator class (Tweepy) **344**
  flatten method **344**
panning a map **364**
payload returned by a Twitter API method **338**
persistent connection 356
Popup class (folium) **368**
project to manager your Twitter apps **336**
pushed tweets from the Twitter Streaming API 356
PyMongo 372

Q
query string 348
query string tutorial (Twitter v2 API) 350

R
rate limit (Twitter API) **334**, 341

recurrent neural network (RNN) 372
relational database
  relational database management system (RDBMS) 372

S

save method of class Map **368**
screen_name property of a User (Twitter) **342**
search method of class Client **347**
search operators in Twitter v2 API query strings 349
sentiment analysis 360
sentiment in tweets 332
set_options function (tweet-preprocessor library) **356**
simple linear regression 372
sorted built-in function 345
standalone Twitter v2 API search operator **349**
Stream class (Tweepy) 360
  filter method **360**
StreamingClient (Tweepy)
  add_rules method **359**
  delete_rules method **359**
  get_rules method **359**
StreamingClient class (Tweepy) **356**
  on_connect method 356, **357**
  on_response method 356, 358
StreamResponse (Tweepy) **356**, 358
StreamRule (Tweepy) **356**, 359, 360

T

time series 372
timeline (Twitter) **343**
trending topics (Twitter) 333, **351**
Trends API (Twitter v1.1 APIs) **334**
trends_place method of class API **352**
Tweepy
  API class **351**
  OAuth2BearerHandler class **351**
  StreamResponse **356**, 358
  StreamRule **356**, 359, 360
Tweepy library **334**, 339
  AsyncStreamingClient class **360**
  Client class **341**
  install 339, 355
  Paginator **344**
  Stream class 360
  StreamingClient class **356**
  wait_on_rate_limit 341
tweepy module 341
tweepy.Response object **342**
tweet JSON object 337
tweet JSON object (Twitter) 338
tweet_fields argument for filtering tweets 366
tweet-preprocessor library **355**
  set_options function **356**
Tweets API (Twitter v2 APIs) **334**
Twitter 333
  API key **336**
  API key secret **336**
  bearer token **336**, 341
  data mining 332
  history 333
  rate limits **334**
  Streaming API 356
  timeline **343**
  trending topics **351**
  tweet JSON object 338
  user JSON object 338
Twitter API 334
  app rate limit 334
  Consumer API keys **336**
  description of a user account **343**
  id of a user account **342**
  name of a user account **342**
  payload returned by a method **338**
  public_metrics of a user account **343**
  rate limit **334**, 341
  Trends API **334**
  user rate limit 334
  username of a user account **342**
Twitter API credentials **336**
Twitter developer account level
  Academic Research 335, 349
  Elevated 335, 336, 349, 351, 356
  Essentials 335, 349, 356
Twitter search operators 349
Twitter Trends API 351
Twitter v1.1 APIs
  Trends API **334**
Twitter v2 API
  /2/tweets/search/recent method **347**
  /2/users/:id/followers method **344**
  /2/users/:id/following method **345**
  /2/users/:id/timelines/reverse_chronological method **347**

/2/users/:id/tweets method **346**
/2/users/by/username/:username method **342**
conjunction-required search operator **349**
expansions 348
expansions in a method call 338
fields in a method call 338
query string tutorial 350
search operators 349
standalone search operator **349**
stream rule **356**, 359, 360
user fields 342
Twitter v2 API expansion 'author_id' 366
Twitter v2 APIs
app (for managing credentials) **336**
project to manage your Twitter apps **336**
Tweets API **334**
Users API **334**
Twittersphere **333**
Twitterverse **333**

U

United States
geographic center 367
user fields in a Twitter response 342
user JSON object 348, 358
user rate limit (Twitter API) 334
user_fields argument for filtering tweets 366
userJSON object (Twitter) 338
Users API (Twitter v2 APIs) **334**

V

visualization
Folium **340**

W

wait_on_rate_limit (Tweepy) 341
web service **334**
endpoint **334**
Webb Space Telescope 348
WOEID (Yahoo! Where on Earth ID) **352**, 352

Y

Yahoo! Where on Earth ID (WOEID) **352**, 352

Z

zoom a map 364