tweepy Documentation

Release 3.8.0

Joshua Roesslein

Aug 01, 2019
## Contents

1 Getting started
- 1.1 Introduction ........................................ 3
- 1.2 Hello Tweepy ........................................ 3
- 1.3 API .................................................. 3
- 1.4 Models .............................................. 4

2 Authentication Tutorial
- 2.1 Introduction ........................................ 5
- 2.2 OAuth 1a Authentication ............................ 5
- 2.3 OAuth 2 Authentication ............................ 7

3 Code Snippets
- 3.1 Introduction ........................................ 9
- 3.2 OAuth ............................................... 9
- 3.3 Pagination ......................................... 9
- 3.4 FollowAll .......................................... 10
- 3.5 Handling the rate limit using cursors .......... 10

4 Cursor Tutorial
- 4.1 Introduction ........................................ 11
- 4.2 Old way vs Cursor way .......................... 11
- 4.3 Passing parameters into the API method .......... 12
- 4.4 Items or Pages .................................... 12
- 4.5 Limits .............................................. 12

5 Extended Tweets
- 5.1 Introduction ........................................ 13
- 5.2 Standard API methods ............................ 13
- 5.3 Streaming .......................................... 14
- 5.4 Handling Retweets ................................ 14
- 5.5 Examples .......................................... 15

6 Streaming With Tweepy
- 6.1 Summary ............................................ 17
- 6.2 Step 1: Creating a StreamListener ............. 18
- 6.3 Step 2: Creating a Stream ....................... 18
- 6.4 Step 3: Starting a Stream ....................... 18
1.1 Introduction

If you are new to Tweepy, this is the place to begin. The goal of this tutorial is to get you set-up and rolling with Tweepy. We won’t go into too much detail here, just some important basics.

1.2 Hello Tweepy

```python
import tweepy
auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
auth.set_access_token(access_token, access_token_secret)
api = tweepy.API(auth)
public_tweets = api.home_timeline()
for tweet in public_tweets:
    print(tweet.text)
```

This example will download your home timeline tweets and print each one of their texts to the console. Twitter requires all requests to use OAuth for authentication. The Authentication Tutorial goes into more details about authentication.

1.3 API

The API class provides access to the entire twitter RESTful API methods. Each method can accept various parameters and return responses. For more information about these methods please refer to API Reference.
1.4 Models

When we invoke an API method most of the time returned back to us will be a Tweepy model class instance. This will contain the data returned from Twitter which we can then use inside our application. For example the following code returns to us an User model:

```python
# Get the User object for twitter...
user = api.get_user('twitter')
```

Models contain the data and some helper methods which we can then use:

```python
print(user.screen_name)
print(user.followers_count)
for friend in user.friends():
    print(friend.screen_name)
```

For more information about models please see ModelsReference.
2.1 Introduction

Tweepy supports both OAuth 1a (application-user) and OAuth 2 (application-only) authentication. Authentication is handled by the tweepy.AuthHandler class.

2.2 OAuth 1a Authentication

Tweepy tries to make OAuth 1a as painless as possible for you. To begin the process we need to register our client application with Twitter. Create a new application and once you are done you should have your consumer key and secret. Keep these two handy, you’ll need them.

The next step is creating an OAuthHandler instance. Into this we pass our consumer key and secret which was given to us in the previous paragraph:

```python
auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
```

If you have a web application and are using a callback URL that needs to be supplied dynamically you would pass it in like so:

```python
auth = tweepy.OAuthHandler(consumer_key, consumer_secret, callback_url)
```

If the callback URL will not be changing, it is best to just configure it statically on twitter.com when setting up your application’s profile.

Unlike basic auth, we must do the OAuth 1a “dance” before we can start using the API. We must complete the following steps:

1. Get a request token from twitter
2. Redirect user to twitter.com to authorize our application
3. If using a callback, twitter will redirect the user to us. Otherwise the user must manually supply us with the verifier code.

4. Exchange the authorized request token for an access token.

So let’s fetch our request token to begin the dance:

```python
try:
    redirect_url = auth.get_authorization_url()
except tweepy.TweepError:
    print('Error! Failed to get request token.')
```

This call requests the token from twitter and returns to us the authorization URL where the user must be redirect to authorize us. Now if this is a desktop application we can just hang onto our OAuthHandler instance until the user returns back. In a web application we will be using a callback request. So we must store the request token in the session since we will need it inside the callback URL request. Here is a pseudo example of storing the request token in a session:

```python
session.set('request_token', auth.request_token['oauth_token'])
```

So now we can redirect the user to the URL returned to us earlier from the get_authorization_url() method.

If this is a desktop application (or any application not using callbacks) we must query the user for the “verifier code” that twitter will supply them after they authorize us. Inside a web application this verifier value will be supplied in the callback request from twitter as a GET query parameter in the URL.

```python
# Example using callback (web app)
verifier = request.GET.get('oauth_verifier')

# Example w/o callback (desktop)
verifier = raw_input('Verifier:')
```

The final step is exchanging the request token for an access token. The access token is the “key” for opening the Twitter API treasure box. To fetch this token we do the following:

```python
try:
    auth.get_access_token(verifier)
except tweepy.TweepError:
    print('Error! Failed to get access token.')
```

It is a good idea to save the access token for later use. You do not need to re-fetch it each time. Twitter currently does not expire the tokens, so the only time it would ever go invalid is if the user revokes our application access. To store the access token depends on your application. Basically you need to store 2 string values: key and secret:

```python
auth.access_token
auth.access_token_secret
```

You can throw these into a database, file, or where ever you store your data. To re-build an OAuthHandler from this stored access token you would do this:
auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
auth.set_access_token(key, secret)

So now that we have our OAuthHandler equipped with an access token, we are ready for business:

api = tweepy.API(auth)
api.update_status('tweepy + oauth!')

## 2.3 OAuth 2 Authentication

Tweepy also supports OAuth 2 authentication. OAuth 2 is a method of authentication where an application makes API requests without the user context. Use this method if you just need read-only access to public information.

Like OAuth 1a, we first register our client application and acquire a consumer key and secret.

Then we create an AppAuthHandler instance, passing in our consumer key and secret:

auth = tweepy.AppAuthHandler(consumer_key, consumer_secret)

With the bearer token received, we are now ready for business:

api = tweepy.API(auth)
for tweet in tweepy.Cursor(api.search, q='tweepy').items(10):
    print(tweet.text)
CHAPTER 3

Code Snippets

3.1 Introduction

Here are some code snippets to help you out with using Tweepy. Feel free to contribute your own snippets or improve the ones here!

3.2 OAuth

```python
auth = tweepy.OAuthHandler("consumer_key", "consumer_secret")

# Redirect user to Twitter to authorize
redirect_user(auth.get_authorization_url())

# Get access token
auth.get_access_token("verifier_value")

# Construct the API instance
api = tweepy.API(auth)
```

3.3 Pagination

```python
# Iterate through all of the authenticated user's friends
for friend in tweepy.Cursor(api.friends).items():
    # Process the friend here
    process_friend(friend)

# Iterate through the first 200 statuses in the home timeline
for status in tweepy.Cursor(api.home_timeline).items(200):
```

(continues on next page)
3.4 FollowAll

This snippet will follow every follower of the authenticated user.

```python
for follower in tweepy.Cursor(api.followers).items():
    follower.follow()
```

3.5 Handling the rate limit using cursors

Since cursors raise `RateLimitError` in their `next()` method, handling them can be done by wrapping the cursor in an iterator.

Running this snippet will print all users you follow that themselves follow less than 300 people total - to exclude obvious spambots, for example - and will wait for 15 minutes each time it hits the rate limit.

```python
# In this example, the handler is time.sleep(15 * 60), # but you can of course handle it in any way you want.

def limit_handled(cursor):
    while True:
        try:
            yield cursor.next()
        except tweepy.RateLimitError:
            time.sleep(15 * 60)

for follower in limit_handled(tweepy.Cursor(api.followers).items()):
    if follower.friends_count < 300:
        print(follower.screen_name)
```
Cursor Tutorial

This tutorial describes details on pagination with Cursor objects.

### 4.1 Introduction

We use pagination a lot in Twitter API development. Iterating through timelines, user lists, direct messages, etc. In order to perform pagination, we must supply a page/cursor parameter with each of our requests. The problem here is this requires a lot of boiler plate code just to manage the pagination loop. To help make pagination easier and require less code, Tweepy has the Cursor object.

### 4.2 Old way vs Cursor way

First let’s demonstrate iterating the statuses in the authenticated user’s timeline. Here is how we would do it the “old way” before the Cursor object was introduced:

```python
page = 1
while True:
    statuses = api.user_timeline(page=page)
    if statuses:
        for status in statuses:
            # process status here
            process_status(status)
    else:
        # All done
        break
    page += 1  # next page
```

As you can see, we must manage the “page” parameter manually in our pagination loop. Now here is the version of the code using the Cursor object:
Now that looks much better! Cursor handles all the pagination work for us behind the scenes, so our code can now focus entirely on processing the results.

### 4.3 Passing parameters into the API method

What if you need to pass in parameters to the API method?

```python
api.user_timeline(id="twitter")
```

Since we pass Cursor the callable, we can not pass the parameters directly into the method. Instead we pass the parameters into the Cursor constructor method:

```python
tweepy.Cursor(api.user_timeline, id="twitter")
```

Now Cursor will pass the parameter into the method for us whenever it makes a request.

### 4.4 Items or Pages

So far we have just demonstrated pagination iterating per item. What if instead you want to process per page of results? You would use the `pages()` method:

```python
for page in tweepy.Cursor(api.user_timeline).pages():
    # page is a list of statuses
    process_page(page)
```

### 4.5 Limits

What if you only want n items or pages returned? You pass into the `items()` or `pages()` methods the limit you want to impose.

```python
# Only iterate through the first 200 statuses
for status in tweepy.Cursor(api.user_timeline).items(200):
    process_status(status)

# Only iterate through the first 3 pages
for page in tweepy.Cursor(api.user_timeline).pages(3):
    process_page(page)
```
CHAPTER 5

Extended Tweets

This supplements Twitter’s Tweet updates documentation.

5.1 Introduction

On May 24, 2016, Twitter announced changes to the way that replies and URLs are handled and published plans around support for these changes in the Twitter API and initial technical documentation describing the updates to Tweet objects and API options.¹ On September 26, 2017, Twitter started testing 280 characters for certain languages,² and on November 7, 2017, announced that the character limit was being expanded for Tweets in languages where cramming was an issue.³

5.2 Standard API methods

Any `tweepy.API` method that returns a `Status` object accepts a new `tweet_mode` parameter. Valid values for this parameter are `compat` and `extended`, which give compatibility mode and extended mode, respectively. The default mode (if no parameter is provided) is compatibility mode.

5.2.1 Compatibility mode

By default, using compatibility mode, the `text` attribute of Status objects returned by `tweepy.API` methods is truncated to 140 characters, as needed. When this truncation occurs, the `truncated` attribute of the Status object will be `True`, and only entities that are fully contained within the available 140 characters range will be included in the `entities` attribute. It will also be discernible that the `text` attribute of the Status object is truncated as it will be suffixed with an ellipsis character, a space, and a shortened self-permalink URL to the Tweet.

---

² [https://twittercommunity.com/t/testing-280-characters-for-certain-languages/94126](https://twittercommunity.com/t/testing-280-characters-for-certain-languages/94126)
5.2.2 Extended mode

When using extended mode, the text attribute of Status objects returned by `tweepy.API` methods is replaced by a full_text attribute, which contains the entire untruncated text of the Tweet. The truncated attribute of the Status object will be False, and the entities attribute will contain all entities. Additionally, the Status object will have a display_text_range attribute, an array of two Unicode code point indices, identifying the inclusive start and exclusive end of the displayable content of the Tweet.

5.3 Streaming

By default, the Status objects from streams may contain an extended_tweet attribute representing the equivalent field in the raw data/payload for the Tweet. This attribute/field will only exist for extended Tweets, containing a dictionary of sub-fields. The full_text sub-field/key of this dictionary will contain the full, untruncated text of the Tweet, and the entities sub-field/key will contain the full set of entities. If there are extended entities, the extended_entities sub-field/key will contain the full set of those. Additionally, the display_text_range sub-field/key will contain an array of two Unicode code point indices, identifying the inclusive start and exclusive end of the displayable content of the Tweet.

5.4 Handling Retweets

When using extended mode with a Retweet, the full_text attribute of the Status object may be truncated with an ellipsis character instead of containing the full text of the Retweet. However, since the retweeted_status attribute (of a Status object that is a Retweet) is itself a Status object, the full_text attribute of the Retweeted Status object can be used instead.

This also applies similarly to Status objects/payloads that are Retweets from streams. The dictionary from the extended_tweet attribute/field contains a full_text sub-field/key that may be truncated with an ellipsis character. Instead, the extended_tweet attribute/field of the Retweeted Status (from the retweeted_status attribute/field) can be used.

5.5 Examples

Given an existing `tweepy.API` object and id for a Tweet, the following can be used to print the full text of the Tweet, or if it’s a Retweet, the full text of the Retweeted Tweet:

```python
status = api.get_status(id, tweet_mode="extended")
try:
    print(status.retweeted_status.full_text)
except AttributeError:  # Not a Retweet
    print(status.full_text)
```

If `status` is a Retweet, `status.full_text` could be truncated.

This Status event handler for a `StreamListener` prints the full text of the Tweet, or if it’s a Retweet, the full text of the Retweeted Tweet:

```python
def on_status(self, status):
    if hasattr(status, "retweeted_status"):  # Check if Retweet
        try:
            print(status.retweeted_status.extended_tweet["full_text"])
```

(continues on next page)
```python
except AttributeError:
    print(status.retweeted_status.text)
else:
    try:
        print(status.extended_tweet["full_text"])  
    except AttributeError:
        print(status.text)
```

If `status` is a Retweet, it will not have an `extended_tweet` attribute, and `status.text` could be truncated.
Tweepy makes it easier to use the twitter streaming api by handling authentication, connection, creating and destroying the session, reading incoming messages, and partially routing messages.

This page aims to help you get started using Twitter streams with Tweepy by offering a first walk through. Some features of Tweepy streaming are not covered here. See streaming.py in the Tweepy source code.

API authorization is required to access Twitter streams. Follow the Authentication Tutorial if you need help with authentication.

6.1 Summary

The Twitter streaming API is used to download twitter messages in real time. It is useful for obtaining a high volume of tweets, or for creating a live feed using a site stream or user stream. See the Twitter Streaming API Documentation.

The streaming api is quite different from the REST api because the REST api is used to pull data from twitter but the streaming api pushes messages to a persistent session. This allows the streaming api to download more data in real time than could be done using the REST API.

In Tweepy, an instance of tweepy.Stream establishes a streaming session and routes messages to StreamListener instance. The on_data method of a stream listener receives all messages and calls functions according to the message type. The default StreamListener can classify most common twitter messages and routes them to appropriately named methods, but these methods are only stubs.

Therefore using the streaming api has three steps.

1. Create a class inheriting from StreamListener
2. Using that class create a Stream object
3. Connect to the Twitter API using the Stream.
6.2 Step 1: Creating a StreamListener

This simple stream listener prints status text. The on_data method of Tweepy’s StreamListener conveniently passes data from statuses to the on_status method. Create class MyStreamListener inheriting from StreamListener and overriding on_status:

```python
import tweepy
#override tweepy.StreamListener to add logic to on_status
class MyStreamListener(tweepy.StreamListener):
    def on_status(self, status):
        print(status.text)
```

6.3 Step 2: Creating a Stream

We need an api to stream. See Authentication Tutorial to learn how to get an api object. Once we have an api and a status listener we can create our stream object:

```python
myStreamListener = MyStreamListener()
myStream = tweepy.Stream(auth = api.auth, listener=myStreamListener)
```

6.4 Step 3: Starting a Stream

A number of twitter streams are available through Tweepy. Most cases will use filter, the user_stream, or the sitestream. For more information on the capabilities and limitations of the different streams see Twitter Streaming API Documentation.

In this example we will use filter to stream all tweets containing the word python. The track parameter is an array of search terms to stream.

```python
myStream.filter(track=['python'])
```

This example shows how to use filter to stream tweets by a specific user. The follow parameter is an array of IDs.

```python
myStream.filter(follow=['2211149702'])
```

An easy way to find a single ID is to use one of the many conversion websites: search for ‘what is my twitter ID’.

6.5 A Few More Pointers

6.5.1 Async Streaming

Streams do not terminate unless the connection is closed, blocking the thread. Tweepy offers a convenient is_async parameter on filter so the stream will run on a new thread. For example

```python
myStream.filter(track=['python'], is_async=True)
```
6.5.2 Handling Errors

When using Twitter’s streaming API one must be careful of the dangers of rate limiting. If clients exceed a limited number of attempts to connect to the streaming API in a window of time, they will receive error 420. The amount of time a client has to wait after receiving error 420 will increase exponentially each time they make a failed attempt.

Tweepy’s Stream Listener passes error codes to an on_error stub. The default implementation returns False for all codes, but we can override it to allow Tweepy to reconnect for some or all codes, using the backoff strategies recommended in the Twitter Streaming API Connecting Documentation.

```python
class MyStreamListener(tweepy.StreamListener):
    def on_error(self, status_code):
        if status_code == 420:
            #returning False in on_error disconnects the stream
            return False
            # returning non-False reconnects the stream, with backoff.
```

For more information on error codes from the Twitter API see Twitter Response Codes Documentation.
This page contains some basic documentation for the Tweepy module.
CHAPTER 8

tweepy.api — Twitter API wrapper

This class provides a wrapper for the API as provided by Twitter. The functions provided in this class are listed below.

Parameters

- **auth_handler** – authentication handler to be used
- **host** – general API host
- **search_host** – search API host
- **cache** – cache backend to use
- **api_root** – general API path root
- **search_root** – search API path root
- **retry_count** – default number of retries to attempt when error occurs
- **retry_delay** – number of seconds to wait between retries
- **retry_errors** – which HTTP status codes to retry
- **timeout** – The maximum amount of time to wait for a response from Twitter
- **parser** – The object to use for parsing the response from Twitter
- **compression** – Whether or not to use GZIP compression for requests
- **wait_on_rate_limit** – Whether or not to automatically wait for rate limits to replenish
- **wait_on_rate_limit_notify** – Whether or not to print a notification when Tweepy is waiting for rate limits to replenish
- **proxy** – The full url to an HTTPS proxy to use for connecting to Twitter.
8.1 Timeline methods

**API.** \texttt{home\_timeline}(\[[\textit{since\_id}], \textit{max\_id}], \textit{count}], \textit{page})

Returns the 20 most recent statuses, including retweets, posted by the authenticating user and that user’s friends. This is the equivalent of /timeline/home on the Web.

**Parameters**

- \textit{since\_id} – Returns only statuses with an ID greater than (that is, more recent than) the specified ID.
- \textit{max\_id} – Returns only statuses with an ID less than (that is, older than) or equal to the specified ID.
- \textit{count} – Specifies the number of statuses to retrieve. Note: there are pagination limits.
- \textit{page} – Specifies the page of results to retrieve.

**Return type** list of \texttt{Status} objects

**API.** \texttt{statuses\_lookup}(\textit{id}, \textit{include\_entities}, \textit{trim\_user}, \textit{map})

Returns full Tweet objects for up to 100 tweets per request, specified by the \textit{id} parameter.

**Parameters**

- \textit{id} – A list of Tweet IDs to lookup, up to 100
- \textit{include\_entities} – A boolean indicating whether or not to include entities in the returned tweets. Defaults to False.
- \textit{trim\_user} – A boolean indicating if user IDs should be provided, instead of full user information. Defaults to False.
- \textit{map} – A boolean indicating whether or not to include tweets that cannot be shown. Defaults to False.

**Return type** list of \texttt{Status} objects

**API.** \texttt{user\_timeline}(\textit{id/user\_id/screen\_name}, \textit{since\_id}, \textit{max\_id}, \textit{count}, \textit{page})

Returns the 20 most recent statuses posted from the authenticating user or the user specified. It’s also possible to request another user’s timeline via the \textit{id} parameter.

**Parameters**

- \textit{id} – Specifies the ID or screen name of the user.
- \textit{user\_id} – Specifies the ID of the user. Helpful for disambiguating when a valid user ID is also a valid screen name.
- \textit{screen\_name} – Specifies the screen name of the user. Helpful for disambiguating when a valid screen name is also a user ID.
- \textit{since\_id} – Returns only statuses with an ID greater than (that is, more recent than) the specified ID.
- \textit{max\_id} – Returns only statuses with an ID less than (that is, older than) or equal to the specified ID.
- \textit{count} – Specifies the number of statuses to retrieve. Note: there are pagination limits.
- \textit{page} – Specifies the page of results to retrieve.

**Return type** list of \texttt{Status} objects
API. retweets_of_me ([since_id], [max_id], [count], [page])
Returns the 20 most recent tweets of the authenticated user that have been retweeted by others.

Parameters
- **since_id** – Returns only statuses with an ID greater than (that is, more recent than) the specified ID.
- **max_id** – Returns only statuses with an ID less than (that is, older than) or equal to the specified ID.
- **count** – Specifies the number of statuses to retrieve.
- **page** – Specifies the page of results to retrieve. Note: there are pagination limits.

Return type list of Status objects

API. mentions_timeline ([since_id], [max_id], [count])
Returns the 20 most recent mentions, including retweets.

Parameters
- **since_id** – Returns only statuses with an ID greater than (that is, more recent than) the specified ID.
- **max_id** – Returns only statuses with an ID less than (that is, older than) or equal to the specified ID.
- **count** – Specifies the number of statuses to retrieve.

Return type list of Status objects

### 8.2 Status methods

API. get_status (id)
Returns a single status specified by the ID parameter.

Parameters **id** – The numerical ID of the status.

Return type Status object

API. update_status (status, in_reply_to_status_id, in_reply_to_status_id_str, auto_populate_reply_metadata, lat, long, source, place_id, display_coordinates, media_ids)
Update the authenticated user’s status. Statuses that are duplicates or too long will be silently ignored.

Parameters
- **status** – The text of your status update.
- **in_reply_to_status_id** – The ID of an existing status that the update is in reply to.
- **in_reply_to_status_id_str** – The ID of an existing status that the update is in reply to (as string).
- **auto_populate_reply_metadata** – Whether to automatically include the @mentions in the status metadata.
- **lat** – The location’s latitude that this tweet refers to.
- **long** – The location’s longitude that this tweet refers to.
- **source** – Source of the update. Only supported by Identi.ca. Twitter ignores this parameter.
• **place_id** – Twitter ID of location which is listed in the Tweet if geolocation is enabled for the user.

• **display_coordinates** – Whether or not to put a pin on the exact coordinates a Tweet has been sent from.

• **media_ids** – A list of media_ids to associate with the Tweet.

**Return type** Status object

```python
API.update_with_media(filename, status, in_reply_to_status_id, auto_populate_reply_metadata, lat, long, source, place_id, file)
```

*Deprecated:* Use `API.media_upload()` instead. Update the authenticated user’s status. Statuses that are duplicates or too long will be silently ignored.

**Parameters**

- **filename** – The filename of the image to upload. This will automatically be opened unless `file` is specified.

- **status** – The text of your status update.

- **in_reply_to_status_id** – The ID of an existing status that the update is in reply to.

- **auto_populate_reply_metadata** – Whether to automatically include the @mentions in the status metadata.

- **lat** – The location’s latitude that this tweet refers to.

- **long** – The location’s longitude that this tweet refers to.

- **source** – Source of the update. Only supported by Identi.ca. Twitter ignores this parameter.

- **place_id** – Twitter ID of location which is listed in the Tweet if geolocation is enabled for the user.

- **file** – A file object, which will be used instead of opening `filename`. `filename` is still required, for MIME type detection and to use as a form field in the POST data

**Return type** Status object

```python
API.destroy_status(id)
```

Destroy the status specified by the id parameter. The authenticated user must be the author of the status to destroy.

**Parameters**

- **id** – The numerical ID of the status.

**Return type** Status object

```python
API.retweet(id)
```

Retweets a tweet. Requires the id of the tweet you are retweeting.

**Parameters**

- **id** – The numerical ID of the status.

**Return type** Status object

```python
API.retweeters(id, cursor, stringify_ids)
```

Returns up to 100 user IDs belonging to users who have retweeted the Tweet specified by the id parameter.

**Parameters**

- **id** – The numerical ID of the status.

- **cursor** – Breaks the results into pages. Provide a value of -1 to begin paging. Provide values as returned to in the response body’s next_cursor and previous_cursor attributes to page back and forth in the list.
• **stringify_ids** – Have ids returned as strings instead.

Return type  list of Integers

API. **retweets** *(id[, count]*)

Returns up to 100 of the first retweets of the given tweet.

Parameters

• **id** – The numerical ID of the status.
• **count** – Specifies the number of retweets to retrieve.

Return type  list of Status objects

API. **unretweet** *(id)*

Untweets a retweeted status. Requires the id of the retweet to unretweet.

Parameters  **id** – The numerical ID of the status.

Return type  Status object

### 8.3 User methods

API. **get_user** *(id/user_id/screen_name)*

Returns information about the specified user.

Parameters

• **id** – Specifies the ID or screen name of the user.
• **user_id** – Specifies the ID of the user. Helpful for disambiguating when a valid user ID is also a valid screen name.
• **screen_name** – Specifies the screen name of the user. Helpful for disambiguating when a valid screen name is also a user ID.

Return type  User object

API. **me** *

Returns the authenticated user’s information.

Return type  User object

API. **friends** *(id/user_id/screen_name [, cursor ][, skip_status ][, include_user_entities ])*

Returns an user’s friends ordered in which they were added 100 at a time. If no user is specified it defaults to the authenticated user.

Parameters

• **id** – Specifies the ID or screen name of the user.
• **user_id** – Specifies the ID of the user. Helpful for disambiguating when a valid user ID is also a valid screen name.
• **screen_name** – Specifies the screen name of the user. Helpful for disambiguating when a valid screen name is also a user ID.
• **cursor** – Breaks the results into pages. Provide a value of -1 to begin paging. Provide values as returned to in the response body’s next_cursor and previous_cursor attributes to page back and forth in the list.
• **count** – Specifies the number of statuses to retrieve.
• **skip_status** – When set to either true, t or 1 statuses will not be included in the returned user objects. Defaults to false.

• **include_user_entities** – The user object entities node will not be included when set to false. Defaults to true.

**Return type** list of User objects

### API.followers(id/screen_name/user_id, cursor)

Returns a user’s followers ordered in which they were added. If no user is specified by id/screen name, it defaults to the authenticated user.

**Parameters**

• **id** – Specifies the ID or screen name of the user.

• **user_id** – Specifies the ID of the user. Helpful for disambiguating when a valid user ID is also a valid screen name.

• **screen_name** – Specifies the screen name of the user. Helpful for disambiguating when a valid screen name is also a user ID.

• **cursor** – Breaks the results into pages. Provide a value of -1 to begin paging. Provide values as returned to in the response body’s next_cursor and previous_cursor attributes to page back and forth in the list.

• **count** – Specifies the number of statuses to retrieve.

• **skip_status** – When set to either true, t or 1 statuses will not be included in the returned user objects. Defaults to false.

• **include_user_entities** – The user object entities node will not be included when set to false. Defaults to true.

**Return type** list of User objects

### API.search_users(q, count, page)

Run a search for users similar to Find People button on Twitter.com; the same results returned by people search on Twitter.com will be returned by using this API (about being listed in the People Search). It is only possible to retrieve the first 1000 matches from this API.

**Parameters**

• **q** – The query to run against people search.

• **count** – Specifies the number of statuses to retrieve. May not be greater than 20.

• **page** – Specifies the page of results to retrieve. Note: there are pagination limits.

**Return type** list of User objects

### 8.4 Direct Message Methods

**API.get_direct_message(id, full_text)**

Returns a specific direct message.

**Parameters**

• **id** – id

• **full_text** – A boolean indicating whether or not the full text of a message should be returned. If False the message text returned will be truncated to 140 chars. Defaults to False.
Returns all Direct Message events (both sent and received) within the last 30 days. Sorted in reverse-chronological order.

**Parameters**

- **count** – Specifies the number of statuses to retrieve.
- **cursor** – Breaks the results into pages. Provide a value of -1 to begin paging. Provide values as returned to in the response body’s next_cursor and previous_cursor attributes to page back and forth in the list.

**Return type** list of DirectMessage objects

Sends a new direct message to the specified user from the authenticating user.

**Parameters**

- **recipient_id** – The ID of the user who should receive the direct message.
- **text** – The text of your Direct Message. Max length of 10,000 characters.
- **quick_reply_type** – The Quick Reply type to present to the user:
  - options - Array of Options objects (20 max).
  - text_input - Text Input object.
  - location - Location object.
- **attachment_type** – The attachment type. Can be media or location.
- **attachment_media_id** – A media id to associate with the message. A Direct Message may only reference a single media_id.

**Return type** DirectMessage object

Deletes the direct message specified in the required ID parameter. The authenticating user must be the recipient of the specified direct message. Direct Messages are only removed from the interface of the user context provided. Other members of the conversation can still access the Direct Messages.

**Parameters** **id** – The id of the Direct Message that should be deleted.

**Return type** None

### 8.5 Friendship Methods

Create a new friendship with the specified user (aka follow).

**Parameters**

- **id** – Specifies the ID or screen name of the user.
- **screen_name** – Specifies the screen name of the user. Helpful for disambiguating when a valid screen name is also a user ID.
- **user_id** – Specifies the ID of the user. Helpful for disambiguating when a valid user ID is also a valid screen name.
• **follow** – Enable notifications for the target user in addition to becoming friends.

**Return type** User object

API. **destroy_friendship** *(id/screen_name/user_id)*

Destroy a friendship with the specified user (aka unfollow).

**Parameters**

- **id** – Specifies the ID or screen name of the user.
- **screen_name** – Specifies the screen name of the user. Helpful for disambiguating when a valid screen name is also a user ID.
- **user_id** – Specifies the ID of the user. Helpful for disambiguating when a valid user ID is also a valid screen name.

**Return type** User object

API. **show_friendship** *(source_id/source_screen_name, target_id/target_screen_name)*

Returns detailed information about the relationship between two users.

**Parameters**

- **source_id** – The user_id of the subject user.
- **source_screen_name** – The screen_name of the subject user.
- **target_id** – The user_id of the target user.
- **target_screen_name** – The screen_name of the target user.

**Return type** Friendship object

API. **friends_ids** *(id/screen_name/user_id[, cursor]*)

Returns an array containing the IDs of users being followed by the specified user.

**Parameters**

- **id** – Specifies the ID or screen name of the user.
- **screen_name** – Specifies the screen name of the user. Helpful for disambiguating when a valid screen name is also a user ID.
- **user_id** – Specifies the ID of the user. Helpful for disambiguating when a valid user ID is also a valid screen name.
- **cursor** – Breaks the results into pages. Provide a value of -1 to begin paging. Provide values as returned to in the response body’s next_cursor and previous_cursor attributes to page back and forth in the list.

**Return type** list of Integers

API. **followers_ids** *(id/screen_name/user_id)*

Returns an array containing the IDs of users following the specified user.

**Parameters**

- **id** – Specifies the ID or screen name of the user.
- **screen_name** – Specifies the screen name of the user. Helpful for disambiguating when a valid screen name is also a user ID.
- **user_id** – Specifies the ID of the user. Helpful for disambiguating when a valid user ID is also a valid screen name.
• **cursor** – Breaks the results into pages. Provide a value of -1 to begin paging. Provide values as returned to in the response body’s nextCursor and previousCursor attributes to page back and forth in the list.

**Return type**: list of Integers

### 8.6 Account Methods

**API.**`verify_credentials()`

Verify the supplied user credentials are valid.

**Return type**: User object if credentials are valid, otherwise False

**API.**`rate_limit_status()`

Returns the remaining number of API requests available to the requesting user before the API limit is reached for the current hour. Calls to rate_limit_status do not count against the rate limit. If authentication credentials are provided, the rate limit status for the authenticating user is returned. Otherwise, the rate limit status for the requester’s IP address is returned.

**Return type**: JSON object

**API.**`update_profile_image`(filename)

Update the authenticating user’s profile image. Valid formats: GIF, JPG, or PNG

**Parameters**: filename – local path to image file to upload. Not a remote URL!

**Return type**: User object

**API.**`update_profile_background_image`(filename)

Update authenticating user’s background image. Valid formats: GIF, JPG, or PNG

**Parameters**: filename – local path to image file to upload. Not a remote URL!

**Return type**: User object

**API.**`update_profile`([name], [url], [location], [description])

Sets values that users are able to set under the “Account” tab of their settings page.

**Parameters**

• **name** – Maximum of 20 characters

• **url** – Maximum of 100 characters. Will be prepended with “http://” if not present

• **location** – Maximum of 30 characters

• **description** – Maximum of 160 characters

**Return type**: User object

### 8.7 Favorite Methods

**API.**`favorites`([id], [page])

Returns the favorite statuses for the authenticating user or user specified by the ID parameter.

**Parameters**

• **id** – The ID or screen name of the user to request favorites

• **page** – Specifies the page of results to retrieve. Note: there are pagination limits.
Return type  list of Status objects

API. **create_favorite** (**id**)  
Favors the status specified in the ID parameter as the authenticating user.

**Parameters**  
id – The numerical ID of the status.

**Return type**  Status object

API. **destroy_favorite** (**id**)  
Un-favors the status specified in the ID parameter as the authenticating user.

**Parameters**  
id – The numerical ID of the status.

**Return type**  Status object

### 8.8 Block Methods

**API. create_block** (**id/screen_name/user_id**)  
Blocks the user specified in the ID parameter as the authenticating user. Destroys a friendship to the blocked user if it exists.

**Parameters**

- **id** – Specifies the ID or screen name of the user.
- **screen_name** – Specifies the screen name of the user. Helpful for disambiguating when a valid screen name is also a user ID.
- **user_id** – Specifies the ID of the user. Helpful for disambiguating when a valid user ID is also a valid screen name.

**Return type**  User object

**API. destroy_block** (**id/screen_name/user_id**)  
Un-blocks the user specified in the ID parameter for the authenticating user.

**Parameters**

- **id** – Specifies the ID or screen name of the user.
- **screen_name** – Specifies the screen name of the user. Helpful for disambiguating when a valid screen name is also a user ID.
- **user_id** – Specifies the ID of the user. Helpful for disambiguating when a valid user ID is also a valid screen name.

**Return type**  User object

**API. blocks** ([**page**])  
Returns an array of user objects that the authenticating user is blocking.

**Parameters**  
page – Specifies the page of results to retrieve. Note: there are pagination limits.

**Return type**  list of User objects

**API. blocks_ids** ([**cursor**])  
Returns an array of numeric user ids the authenticating user is blocking.

**Parameters**  
cursor – Breaks the results into pages. Provide a value of -1 to begin paging. Provide values as returned to in the response body’s next_cursor and previous_cursor attributes to page back and forth in the list.

**Return type**  list of Integers
8.9 Mute Methods

**API.** `create_mute` *(id/screen_name/user_id)*

Mutes the user specified in the ID parameter for the authenticating user.

**Parameters**

- `id` – Specifies the ID or screen name of the user.
- `screen_name` – Specifies the screen name of the user. Helpful for disambiguating when a valid screen name is also a user ID.
- `user_id` – Specifies the ID of the user. Helpful for disambiguating when a valid user ID is also a valid screen name.

**Return type** User object

**API.** `destroy_mute` *(id/screen_name/user_id)*

Un-mutes the user specified in the ID parameter for the authenticating user.

**Parameters**

- `id` – Specifies the ID or screen name of the user.
- `screen_name` – Specifies the screen name of the user. Helpful for disambiguating when a valid screen name is also a user ID.
- `user_id` – Specifies the ID of the user. Helpful for disambiguating when a valid user ID is also a valid screen name.

**Return type** User object

**API.** `mutes` *(cursor*, include_entities*, skip_status*)

Returns an array of user objects the authenticating user has muted.

**Parameters**

- `cursor` – Breaks the results into pages. Provide a value of -1 to begin paging. Provide values as returned to in the response body’s next_cursor and previous_cursor attributes to page back and forth in the list.
- `include_entities` – The entities node will not be included when set to false. Defaults to true.
- `skip_status` – When set to either true, t or 1 statuses will not be included in the returned user objects. Defaults to false.

**Return type** list of User objects

**API.** `mutes_ids` *(cursor*)

Returns an array of numeric user ids the authenticating user has muted.

**Parameters**

`cursor` – Breaks the results into pages. Provide a value of -1 to begin paging. Provide values as returned to in the response body’s next_cursor and previous_cursor attributes to page back and forth in the list.

**Return type** list of Integers
8.10 Spam Reporting Methods

API . report_spam (id/screen_name/user_id[, perform_block])

The user specified in the id is blocked by the authenticated user and reported as a spammer.

Parameters

- **id** – Specifies the ID or screen name of the user.
- **screen_name** – Specifies the screen name of the user. Helpful for disambiguating when a valid screen name is also a user ID.
- **user_id** – Specifies the ID of the user. Helpful for disambiguating when a valid user ID is also a valid screen name.
- **perform_block** – A boolean indicating if the reported account should be blocked. Defaults to True.

Return type User object

8.11 Saved Searches Methods

API . saved_searches ()

Returns the authenticated user’s saved search queries.

Return type list of SavedSearch objects

API . get_saved_search (id)

Retrieve the data for a saved search owned by the authenticating user specified by the given id.

Parameters **id** – The id of the saved search to be retrieved.

Return type SavedSearch object

API . create_saved_search (query)

Creates a saved search for the authenticated user.

Parameters **query** – The query of the search the user would like to save.

Return type SavedSearch object

API . destroy_saved_search (id)

Destroys a saved search for the authenticated user. The search specified by id must be owned by the authenticating user.

Parameters **id** – The id of the saved search to be deleted.

Return type SavedSearch object

8.12 Help Methods

API . search (q[, geocode][, lang][, locale][, result_type][, count][, until][, since_id][, max_id][, include_entities])

Returns tweets that match a specified query.

Parameters

- **q** – the search query string of 500 characters maximum, including operators. Queries may additionally be limited by complexity.
• **geocode** – Returns tweets by users located within a given latitude/longitude. The location is preferentially taking from the Geotagging API, but will fall back to their Twitter profile. The parameter value is specified by “latitude, longitude, radius”, where radius units must be specified as either “mi” (miles) or “km” (kilometers). Note that you cannot use the near operator via the API to geocode arbitrary locations; however you can use this geocode parameter to search near geocodes directly. A maximum of 1,000 distinct “sub-regions” will be considered when using the radius modifier.

• **lang** – Restricts tweets to the given language, given by an ISO 639-1 code. Language detection is best-effort.

• **locale** – Specify the language of the query you are sending (only ja is currently effective). This is intended for language-specific consumers and the default should work in the majority of cases.

• **result_type** – Specifies what type of search results you would prefer to receive. The current default is “mixed.” Valid values include:
  
  – mixed : include both popular and real time results in the response
  – recent : return only the most recent results in the response
  – popular : return only the most popular results in the response

• **count** – The number of tweets to return per page, up to a maximum of 100. Defaults to 15.

• **until** – Returns tweets created before the given date. Date should be formatted as YYYY-MM-DD. Keep in mind that the search index has a 7-day limit. In other words, no tweets will be found for a date older than one week.

• **since_id** – Returns only statuses with an ID greater than (that is, more recent than) the specified ID. There are limits to the number of Tweets which can be accessed through the API. If the limit of Tweets has occurred since the since_id, the since_id will be forced to the oldest ID available.

• **max_id** – Returns only statuses with an ID less than (that is, older than) or equal to the specified ID.

• **include_entities** – The entities node will not be included when set to false. Defaults to true.

**Return type** SearchResults object

### 8.13 List Methods

**API.create_list**(name[, mode][, description])

Creates a new list for the authenticated user. Note that you can create up to 1000 lists per account.

**Parameters**

• **name** – The name of the new list.

• **mode** – Whether your list is public or private. Values can be public or private. Lists are public by default if no mode is specified.

• **description** – The description of the list you are creating.

**Return type** List object

**API.destroy_list**(owner_screen_name/owner_id[, list_id/slug])

Deletes the specified list. The authenticated user must own the list to be able to destroy it.
Parameters

- **owner_screen_name** – The screen name of the user who owns the list being requested by a slug.
- **owner_id** – The user ID of the user who owns the list being requested by a slug.
- **list_id** – The numerical id of the list.
- **slug** – You can identify a list by its slug instead of its numerical id. If you decide to do so, note that you’ll also have to specify the list owner using the owner_id or owner_screen_name parameters.

Return type List object

API.**update_list**(list_id/slug[, name ][, mode ][, description ][, owner_screen_name/owner_id ])

Updates the specified list. The authenticated user must own the list to be able to update it.

Parameters

- **list_id** – The numerical id of the list.
- **slug** – You can identify a list by its slug instead of its numerical id. If you decide to do so, note that you’ll also have to specify the list owner using the owner_id or owner_screen_name parameters.
- **name** – The name for the list.
- **mode** – Whether your list is public or private. Values can be public or private. Lists are public by default if no mode is specified.
- **description** – The description to give the list.
- **owner_screen_name** – The screen name of the user who owns the list being requested by a slug.
- **owner_id** – The user ID of the user who owns the list being requested by a slug.

Return type List object

API.**lists_all**(cursor)

List the lists of the specified user. Private lists will be included if the authenticated users is the same as the user who’s lists are being returned.

Parameters cursor – Breaks the results into pages. Provide a value of -1 to begin paging. Provide values as returned to in the response body’s next_cursor and previous_cursor attributes to page back and forth in the list.

Return type list of List objects

API.**lists_memberships**(cursor)

List the lists the specified user has been added to.

Parameters cursor – Breaks the results into pages. Provide a value of -1 to begin paging. Provide values as returned to in the response body’s next_cursor and previous_cursor attributes to page back and forth in the list.

Return type list of List objects

API.**lists_subscriptions**(cursor)

List the lists the specified user follows.

Parameters cursor – Breaks the results into pages. Provide a value of -1 to begin paging. Provide values as returned to in the response body’s next_cursor and previous_cursor attributes to page back and forth in the list.
Return type: list of List objects

API.list_timeline(owner, slug[, since_id, max_id[, count[, page]]])

Show tweet timeline for members of the specified list.

Parameters

- **owner** – the screen name of the owner of the list
- **slug** – You can identify a list by its slug instead of its numerical id. If you decide to do so, note that you’ll also have to specify the list owner using the owner_id or owner_screen_name parameters.
- **since_id** – Returns only statuses with an ID greater than (that is, more recent than) the specified ID.
- **max_id** – Returns only statuses with an ID less than (that is, older than) or equal to the specified ID.
- **count** – Number of results per a page
- **page** – Specifies the page of results to retrieve. Note: there are pagination limits.

Return type: list of Status objects

API.get_list(list_id/slug, owner_id/owner_screen_name)

Returns the specified list. Private lists will only be shown if the authenticated user owns the specified list.

Parameters

- **list_id** – The numerical id of the list.
- **slug** – You can identify a list by its slug instead of its numerical id. If you decide to do so, note that you’ll also have to specify the list owner using the owner_id or owner_screen_name parameters.
- **owner_id** – The user ID of the user who owns the list being requested by a slug.
- **owner_screen_name** – The screen name of the user who owns the list being requested by a slug.

Return type: List object

API.add_list_member(list_id/slug, screen_name/user_id, owner_id/owner_screen_name)

Add a member to a list. The authenticated user must own the list to be able to add members to it. Lists are limited to 5,000 members.

Parameters

- **list_id** – The numerical id of the list.
- **slug** – You can identify a list by its slug instead of its numerical id. If you decide to do so, note that you’ll also have to specify the list owner using the owner_id or owner_screen_name parameters.
- **screen_name** – Specifies the screen name of the user. Helpful for disambiguating when a valid screen name is also a user ID.
- **user_id** – Specifies the ID of the user. Helpful for disambiguating when a valid user ID is also a valid screen name.
- **owner_id** – The user ID of the user who owns the list being requested by a slug.
- **owner_screen_name** – The screen name of the user who owns the list being requested by a slug.
Add up to 100 members to a list. The authenticated user must own the list to be able to add members to it. Lists are limited to 5,000 members.

Parameters

- **list_id** – The numerical id of the list.
- **slug** – You can identify a list by its slug instead of its numerical id. If you decide to do so, note that you’ll also have to specify the list owner using the owner_id or owner_screen_name parameters.
- **screen_name** – A comma separated list of screen names, up to 100 are allowed in a single request
- **user_id** – A comma separated list of user IDs, up to 100 are allowed in a single request
- **owner_id** – The user ID of the user who owns the list being requested by a slug.
- **owner_screen_name** – The screen name of the user who owns the list being requested by a slug.

Remove the specified member from the list. The authenticated user must be the list’s owner to remove members from the list.

Parameters

- **slug** – You can identify a list by its slug instead of its numerical id. If you decide to do so, note that you’ll also have to specify the list owner using the owner_id or owner_screen_name parameters.
- **id** – the ID of the user to remove as a member

Remove up to 100 members from a list. The authenticated user must own the list to be able to remove members from it. Lists are limited to 5,000 members.

Parameters

- **list_id** – The numerical id of the list.
- **slug** – You can identify a list by its slug instead of its numerical id. If you decide to do so, note that you’ll also have to specify the list owner using the owner_id or owner_screen_name parameters.
- **screen_name** – A comma separated list of screen names, up to 100 are allowed in a single request
- **user_id** – A comma separated list of user IDs, up to 100 are allowed in a single request
- **owner_id** – The user ID of the user who owns the list being requested by a slug.
- **owner_screen_name** – The screen name of the user who owns the list being requested by a slug.

Returns the members of the specified list.
Parameters

- **list_id** – The numerical id of the list.
- **slug** – You can identify a list by its slug instead of its numerical id. If you decide to do so, note that you’ll also have to specify the list owner using the owner_id or owner_screen_name parameters.
- **owner_id** – The user ID of the user who owns the list being requested by a slug.
- **owner_screen_name** – The screen name of the user who owns the list being requested by a slug.
- **cursor** – Breaks the results into pages. Provide a value of -1 to begin paging. Provide values as returned to in the response body’s next_cursor and previous_cursor attributes to page back and forth in the list.

Return type  list of User objects

API.**show_list_member**(list_id/slug, screen_name/user_id[, owner_id/owner_screen_name])

Check if the specified user is a member of the specified list.

Parameters

- **list_id** – The numerical id of the list.
- **slug** – You can identify a list by its slug instead of its numerical id. If you decide to do so, note that you’ll also have to specify the list owner using the owner_id or owner_screen_name parameters.
- **screen_name** – Specifies the screen name of the user. Helpful for disambiguating when a valid screen name is also a user ID.
- **user_id** – Specifies the ID of the user. Helpful for disambiguating when a valid user ID is also a valid screen name.
- **owner_id** – The user ID of the user who owns the list being requested by a slug.
- **owner_screen_name** – The screen name of the user who owns the list being requested by a slug.

Return type  User object if user is a member of list

API.**subscribe_list**(owner, slug)

Make the authenticated user follow the specified list.

Parameters

- **owner** – the screen name of the owner of the list
- **slug** – You can identify a list by its slug instead of its numerical id. If you decide to do so, note that you’ll also have to specify the list owner using the owner_id or owner_screen_name parameters.

Return type  List object

API.**unsubscribe_list**(owner, slug)

Unsubscribes the authenticated user form the specified list.

Parameters

- **owner** – the screen name of the owner of the list
- **slug** – You can identify a list by its slug instead of its numerical id. If you decide to do so, note that you’ll also have to specify the list owner using the owner_id or owner_screen_name parameters.
Return type  List object

API.list_subscribers(owner, slug[, cursor])

Returns the subscribers of the specified list.

Parameters

• **owner** – the screen name of the owner of the list

• **slug** – You can identify a list by its slug instead of its numerical id. If you decide to do so, note that you’ll also have to specify the list owner using the owner_id or owner_screen_name parameters.

• **cursor** – Breaks the results into pages. Provide a value of -1 to begin paging. Provide values as returned to in the response body’s next_cursor and previous_cursor attributes to page back and forth in the list.

Return type  list of User objects

API.show_list_subscriber(list_id/slug, screen_name/user_id[, owner_id/owner_screen_name])

Check if the specified user is a subscriber of the specified list.

Parameters

• **list_id** – The numerical id of the list.

• **slug** – You can identify a list by its slug instead of its numerical id. If you decide to do so, note that you’ll also have to specify the list owner using the owner_id or owner_screen_name parameters.

• **screen_name** – Specifies the screen name of the user. Helpful for disambiguating when a valid screen name is also a user ID.

• **user_id** – Specifies the ID of the user. Helpful for disambiguating when a valid user ID is also a valid screen name.

• **owner_id** – The user ID of the user who owns the list being requested by a slug.

• **owner_screen_name** – The screen name of the user who owns the list being requested by a slug.

Return type  User object if user is subscribed to list

### 8.14 Trends Methods

**API.trends_available()**

Returns the locations that Twitter has trending topic information for. The response is an array of “locations” that encode the location’s WOEID (a Yahoo! Where On Earth ID) and some other human-readable information such as a canonical name and country the location belongs in.

Return type  JSON object

**API.trends_place(id[, exclude])**

Returns the top 50 trending topics for a specific WOEID, if trending information is available for it.

The response is an array of “trend” objects that encode the name of the trending topic, the query parameter that can be used to search for the topic on Twitter Search, and the Twitter Search URL.

This information is cached for 5 minutes. Requesting more frequently than that will not return any more data, and will count against your rate limit usage.

The tweet_volume for the last 24 hours is also returned for many trends if this is available.
Parameters

- **id** – The Yahoo! Where On Earth ID of the location to return trending information for. Global information is available by using 1 as the WOEID.
- **exclude** – Setting this equal to hashtags will remove all hashtags from the trends list.

Return type **JSON object**

API `.trends_closest(lat, long)`

Returns the locations that Twitter has trending topic information for, closest to a specified location.

The response is an array of “locations” that encode the location’s WOEID and some other human-readable information such as a canonical name and country the location belongs in.

A WOEID is a Yahoo! Where On Earth ID.

Parameters

- **lat** – If provided with a long parameter the available trend locations will be sorted by distance, nearest to furthest, to the co-ordinate pair. The valid ranges for longitude is -180.0 to +180.0 (West is negative, East is positive) inclusive.
- **long** – If provided with a lat parameter the available trend locations will be sorted by distance, nearest to furthest, to the co-ordinate pair. The valid ranges for longitude is -180.0 to +180.0 (West is negative, East is positive) inclusive.

Return type **JSON object**

8.15 Geo Methods

API `.reverse_geocode([lat][, long][, accuracy][, granularity][, max_results])`

Given a latitude and longitude, looks for places (cities and neighbourhoods) whose IDs can be specified in a call to `update_status()` to appear as the name of the location. This call provides a detailed response about the location in question; the `nearby_places()` function should be preferred for getting a list of places nearby without great detail.

Parameters

- **lat** – The location’s latitude.
- **long** – The location’s longitude.
- **accuracy** – Specify the “region” in which to search, such as a number (then this is a radius in meters, but it can also take a string that is suffixed with ft to specify feet). If this is not passed in, then it is assumed to be 0m
- **granularity** – Assumed to be ‘neighborhood’ by default; can also be ‘city’.
- **max_results** – A hint as to the maximum number of results to return. This is only a guideline, which may not be adhered to.

API `.reverse_geocode([lat][, long][, ip][, accuracy][, granularity][, max_results])`

Given a latitude and longitude, looks for nearby places (cities and neighbourhoods) whose IDs can be specified in a call to `update_status()` to appear as the name of the location. This call provides a detailed response about the location in question; the `nearby_places()` function should be preferred for getting a list of places nearby without great detail.

Parameters

- **lat** – The location’s latitude.
• **long** – The location’s longitude.

• **ip** – The location’s IP address. Twitter will attempt to geolocate using the IP address.

• **accuracy** – Specify the “region” in which to search, such as a number (then this is a radius in meters, but it can also take a string that is suffixed with ft to specify feet). If this is not passed in, then it is assumed to be 0m

• **granularity** – Assumed to be ‘neighborhood’ by default; can also be ‘city’.

• **max_results** – A hint as to the maximum number of results to return. This is only a guideline, which may not be adhered to.

```python
API.geo_id(id)
```

Given *id* of a place, provide more details about that place.

**Parameters**  
*id* – Valid Twitter ID of a location.

### 8.16 Utility methods

**API.configuration()**

Returns the current configuration used by Twitter including twitter.com slugs which are not usernames, maximum photo resolutions, and t.co shortened URL length. It is recommended applications request this endpoint when they are loaded, but no more than once a day.

### 8.17 Media methods

**API.media_upload()**

Uploads images to twitter and returns a *media_id*.

**Parameters**

• **media** – The raw binary file content being uploaded. Cannot be used with *media_data*.

• **media_data** – The base64-encoded file content being uploaded. Cannot be used with *media*.

• **additional_owners** – A comma-separated list of user IDs to set as additional owners allowed to use the returned *media_id* in Tweets or Cards. Up to 100 additional owners may be specified.
The exceptions are available in the `tweepy` module directly, which means `tweepy.error` itself does not need to be imported. For example, `tweepy.error.TweepError` is available as `tweepy.TweepError`.

**exception TweepError**

The main exception Tweepy uses. Is raised for a number of things.

When a `TweepError` is raised due to an error Twitter responded with, the error code (as described in the API documentation) can be accessed at `TweepError.response.text`. Note, however, that `TweepError` also may be raised with other things as message (for example plain error reason strings).

**exception RateLimitError**

Is raised when an API method fails due to hitting Twitter’s rate limit. Makes for easy handling of the rate limit specifically.

Inherits from `TweepError`, so except `TweepError` will catch a `RateLimitError` too.
Indices and tables

• genindex
• search
<table>
<thead>
<tr>
<th>Index</th>
<th></th>
</tr>
</thead>
</table>
| **A** | add_list_member() *(API method)*, 37  
| | add_list_members() *(API method)*, 38  
| | API *(built-in class)*, 23  
| **B** | blocks() *(API method)*, 32  
| | blocks_ids() *(API method)*, 32  
| **C** | configuration() *(API method)*, 42  
| | create_block() *(API method)*, 32  
| | create_favorite() *(API method)*, 32  
| | create_friendship() *(API method)*, 29  
| | create_list() *(API method)*, 35  
| | create_mute() *(API method)*, 33  
| | create_saved_search() *(API method)*, 34  
| | destroy_block() *(API method)*, 32  
| | destroy_direct_message() *(API method)*, 29  
| | destroy_favorite() *(API method)*, 32  
| | destroy_friendship() *(API method)*, 30  
| | destroy_list() *(API method)*, 35  
| | destroy_mute() *(API method)*, 33  
| | destroy_saved_search() *(API method)*, 34  
| | destroy_status() *(API method)*, 26  
| **D** | favorites() *(API method)*, 31  
| | followers() *(API method)*, 28  
| | followers_ids() *(API method)*, 30  
| | friends() *(API method)*, 27  
| | friends_ids() *(API method)*, 30  
| **G** | geo_id() *(API method)*, 42  
| | get_direct_message() *(API method)*, 28  
| | get_list() *(API method)*, 37  
| | get_saved_search() *(API method)*, 34  
| | get_status() *(API method)*, 25  
| | get_user() *(API method)*, 27  
| **H** | home_timeline() *(API method)*, 24  
| **L** | list_direct_messages() *(API method)*, 29  
| | list_members() *(API method)*, 38  
| | list_subscribers() *(API method)*, 40  
| | list_timeline() *(API method)*, 37  
| | lists_all() *(API method)*, 36  
| | lists_memberships() *(API method)*, 36  
| | lists_subscriptions() *(API method)*, 36  
| **M** | me() *(API method)*, 27  
| | media_upload() *(API method)*, 42  
| | mentions_timeline() *(API method)*, 25  
| | mutes() *(API method)*, 33  
| | mutes_ids() *(API method)*, 33  
| **R** | rate_limit_status() *(API method)*, 31  
| | rate_limit_error() *Exception*, 43  
| | remove_list_member() *(API method)*, 38  
| | remove_list_members() *(API method)*, 38  
| | report_spam() *(API method)*, 34  
| | retweet() *(API method)*, 26  
| | retweeters() *(API method)*, 26  
| | retweets() *(API method)*, 27  
| | retweets_of_me() *(API method)*, 24  
| | reverse_geocode() *(API method)*, 41  
| **S** | saved_searches() *(API method)*, 34  
| | search() *(API method)*, 34  
| | search_users() *(API method)*, 28 |
send_direct_message() *(API method)*, 29
show_friendship() *(API method)*, 30
show_list_member() *(API method)*, 39
show_list_subscriber() *(API method)*, 40
statuses_lookup() *(API method)*, 24
subscribe_list() *(API method)*, 39

trends_available() *(API method)*, 40
trends_closest() *(API method)*, 41
trends_place() *(API method)*, 40
TweepError, 43

unretweet() *(API method)*, 27
unsubscribe_list() *(API method)*, 39
update_list() *(API method)*, 36
update_profile() *(API method)*, 31
update_profile_background_image() *(API method)*, 31
update_profile_image() *(API method)*, 31
update_status() *(API method)*, 25
update_with_media() *(API method)*, 26
user_timeline() *(API method)*, 24

verify_credentials() *(API method)*, 31