Mastodon.py Documentation

Release 1.4.0

Lorenz Diener

Apr 28, 2019

Contents

1 A note about rate limits **3**

2 A note about pagination **5**

3 Two notes about IDs **7**
3.1 ID unpacking . . . 7

4 Error handling **9**

5 Return values **11**
5.1 User dicts . . . 11
5.2 Toot dicts . . . 12
5.3 Mention dicts . . . 13
5.4 Scheduled toot dicts . . . 13
5.5 Poll dicts . . . 13
5.6 Conversation dicts . . . 14
5.7 Hashtag dicts . . . 14
5.8 Hashtag usage history dicts . . . 14
5.9 Emoji dicts . . . 15
5.10 Application dicts . . . 15
5.11 Relationship dicts . . . 15
5.12 Filter dicts . . . 15
5.13 Notification dicts . . . 16
5.14 Context dicts . . . 16
5.15 List dicts . . . 16
5.16 Media dicts . . . 16
5.17 Card dicts . . . 17
5.18 Search result dicts . . . 18
5.19 Instance dicts . . . 18
5.20 Activity dicts . . . 18
5.21 Report dicts . . . 19
5.22 Push subscription dicts . . . 19
5.23 Push notification dicts . . . 19
5.24 Preference dicts . . . 20

6 App registration and user authentication **21**

7 Versioning 25

8 Reading data: Instances 27

9 Reading data: Timelines 29

10 Reading data: Statuses 31
10.1 Reading data: Scheduled statuses 32
10.2 Reading data: Polls 32

11 Reading data: Notifications 33

12 Reading data: Accounts 35

13 Reading data: Keyword filters 37

14 Reading data: Follow suggestions 39

15 Reading data: Lists 41

16 Reading data: Follows 43

17 Reading data: Favourites 45

18 Reading data: Follow requests 47

19 Reading data: Searching 49

20 Reading data: Mutes and blocks 51

21 Reading data: Reports (REMOVED IN 2.5.0) 53

22 Reading data: Domain blocks 55

23 Reading data: Emoji 57

24 Reading data: Apps 59

25 Reading data: Endorsements 61

26 Reading data: Preferences 63

27 Writing data: Statuses 65
27.1 Writing data: Scheduled statuses 67
27.2 Writing data: Polls 67

28 Writing data: Notifications 69

29 Writing data: Conversations 71

30 Writing data: Accounts 73

31 Writing data: Keyword filters 75

32 Writing data: Follow suggestions 77

33 Writing data: Lists 79

34 Writing data: Follow requests **81**

35 Writing data: Media **83**

36 Writing data: Reports **85**

37 Writing data: Domain blocks **87**

38 Pagination **89**

39 Streaming **91**
39.1 StreamListener 92
39.2 CallbackStreamListener 92

40 Push subscriptions **93**

41 Acknowledgements **95**

Python Module Index **97**

Register your app! This only needs to be done once. Uncomment the code and substitute in your information:

```
from mastodon import Mastodon

'''
Mastodon.create_app(
     'pytooterapp',
     api_base_url = 'https://mastodon.social',
     to_file = 'pytooter_clientcred.secret'
)
'''
```

Then login. This can be done every time, or you can use the persisted information:

```
from mastodon import Mastodon

mastodon = Mastodon(
    client_id = 'pytooter_clientcred.secret',
    api_base_url = 'https://mastodon.social'
)
mastodon.log_in(
    'my_login_email@example.com',
    'incrediblygoodpassword',
    to_file = 'pytooter_usercred.secret'
)
```

To post, create an actual API instance:

```
from mastodon import Mastodon

mastodon = Mastodon(
    access_token = 'pytooter_usercred.secret',
    api_base_url = 'https://mastodon.social'
)
mastodon.toot('Tooting from python using #mastodonpy !')
```

Mastodon is an ActivityPub and OStatus based twitter-like federated social network node. It has an API that allows you to interact with its every aspect. This is a simple python wrapper for that api, provided as a single python module. By default, it talks to the Mastodon flagship instance, but it can be set to talk to any node running Mastodon by setting *api_base_url* when creating the api object (or creating an app).

Mastodon.py aims to implement the complete public Mastodon API. As of this time, it is feature complete for Mastodon version 2.8.0. Pleromas Mastodon API layer, while not an official target, should also be basically compatible.

CHAPTER 1

A note about rate limits

Mastodons API rate limits per user account. By default, the limit is 300 requests per 5 minute time slot. This can differ from instance to instance and is subject to change. Mastodon.py has three modes for dealing with rate limiting that you can pass to the constructor, "throw", "wait" and "pace", "wait" being the default.

In "throw" mode, Mastodon.py makes no attempt to stick to rate limits. When a request hits the rate limit, it simply throws a *MastodonRateLimitError*. This is for applications that need to handle all rate limiting themselves (i.e. interactive apps), or applications wanting to use Mastodon.py in a multi-threaded context ("wait" and "pace" modes are not thread safe).

Note: Rate limit information is available on the *Mastodon* object for applications that implement their own rate limit handling.

`Mastodon.`**`ratelimit_remaining`**
: Number of requests allowed until the next reset.

`Mastodon.`**`ratelimit_reset`**
: Time at which the rate limit will next be reset, as a POSIX timestamp.

`Mastodon.`**`ratelimit_limit`**
: Total number of requests allowed between resets. Typically 300.

`Mastodon.`**`ratelimit_lastcall`**
: Time at which these values have last been seen and updated, as a POSIX timestamp.

In "wait" mode, once a request hits the rate limit, Mastodon.py will wait until the rate limit resets and then try again, until the request succeeds or an error is encountered. This mode is for applications that would rather just not worry about rate limits much, don't poll the api all that often, and are okay with a call sometimes just taking a while.

In "pace" mode, Mastodon.py will delay each new request after the first one such that, if requests were to continue at the same rate, only a certain fraction (set in the constructor as *ratelimit_pacefactor*) of the rate limit will be used up. The fraction can be (and by default, is) greater than one. If the rate limit is hit, "pace" behaves like "wait". This mode is probably the most advanced one and allows you to just poll in a loop without ever sleeping at all yourself. It is for applications that would rather just pretend there is no such thing as a rate limit and are fine with sometimes not being very interactive.

In addition to the per-user limit, there is a per-IP limit of 7500 requests per 5 minute time slot, and tighter limits on logins. Mastodon.py does not make any effort to respect these.

If your application requires many hits to endpoints that are available without logging in, do consider using Mastodon.py without authenticating to get the full per-IP limit.

CHAPTER 2

A note about pagination

Many of Mastodons API endpoints are paginated. What this means is that if you request data from them, you might not get all the data at once - instead, you might only get the first few results.

All endpoints that are paginated have four parameters: since_id, max_id, min_id and limit. since_id allows you to specify the smallest id you want in the returned data, but you will still always get the newest data, so if there are too many statuses between the newest one and since_id, some will not be returned. min_id, on the other hand, gives you statuses with that minimum id and newer, starting at the given id. max_id, similarly, allows you to specify the largest id you want. By specifying either min_id or max_id (generally, only one, not both) of them you can go through pages forwards and backwards.

limit allows you to specify how many results you would like returned. Note that an instance may choose to return less results than you requested - by default, Mastodon will return no more than 40 statues and no more than 80 accounts no matter how high you set the limit.

The responses returned by paginated endpoints contain a "link" header that specifies which parameters to use to get the next and previous pages. Mastodon.py parses these and stores them (if present) in the first (for the previous page) and last (for the next page) item of the returned list as _pagination_prev and _pagination_next. They are accessible only via attribute-style access. Note that this means that if you want to persist pagination info with your data, you'll have to take care of that manually (or persist objects, not just dicts).

There are convenience functions available for fetching the previous and next page of a paginated request as well as for fetching all pages starting from a first page.

CHAPTER 3

Two notes about IDs

Mastodons API uses IDs in several places: User IDs, Toot IDs, …

While debugging, it might be tempting to copy-paste in IDs from the web interface into your code. This will not work, as the IDs on the web interface and in the URLs are not the same as the IDs used internally in the API, so don't do that.

3.1 ID unpacking

Wherever Mastodon.py expects an ID as a parameter, you can also pass a dict that contains an id - this means that, for example, instead of writing

```
mastodon.status_post("@somebody wow!", in_reply_to_id = toot["id"])
```

you can also just write

```
mastodon.status_post("@somebody wow!", in_reply_to_id = toot)
```

and everything will work as intended.

CHAPTER 4

Error handling

When Mastodon.py encounters an error, it will raise an exception, generally with some text included to tell you what went wrong.

The base class that all mastodon exceptions inherit from is *MastodonError*. If you are only interested in the fact an error was raised somewhere in Mastodon.py, and not the details, this is the exception you can catch.

MastodonIllegalArgumentError is generally a programming problem - you asked the API to do something obviously invalid (i.e. specify a privacy option that does not exist).

MastodonFileNotFoundError and *MastodonNetworkError* are IO errors - could be you specified a wrong URL, could be the internet is down or your hard drive is dying. They inherit from *MastodonIOError*, for easy catching. There is a sub-error of *MastodonNetworkError*, *MastodonReadTimeout*, which is thrown when a streaming API stream times out during reading.

MastodonAPIError is an error returned from the Mastodon instance - the server has decided it can't fullfill your request (i.e. you requested info on a user that does not exist). It is further split into *MastodonNotFoundError* (API returned 404) and *MastodonUnauthorizedError* (API returned 401). Different error codes might exist, but are not currently handled separately.

MastodonMalformedEventError is raised when a streaming API listener receives an invalid event. There have been reports that this can sometimes happen after prolonged operation due to an upstream problem in the requests/urllib libraries.

MastodonRatelimitError is raised when you hit an API rate limit. You should try again after a while (see the rate limiting section above).

MastodonServerError is raised when the server throws an internal error, likely due to server misconfiguration.

MastodonVersionError is raised when a version check for an API call fails.

CHAPTER 5

Return values

Unless otherwise specified, all data is returned as python dictionaries, matching the JSON format used by the API. Dates returned by the API are in ISO 8601 format and are parsed into python datetime objects.

To make access easier, the dictionaries returned are wrapped by a class that adds read-only attributes for all dict values - this means that, for example, instead of writing

```
description = mastodon.account_verify_credentials()["source"]["note"]
```

you can also just write

```
description = mastodon.account_verify_credentials().source.note
```

and everything will work as intended. The class used for this is exposed as *AttribAccessDict*.

5.1 User dicts

```
mastodon.account(<numerical id>)
# Returns the following dictionary:
{
    'id': # Same as <numerical id>
    'username': # The username (what you @ them with)
    'acct': # The user's account name as username@domain (@domain omitted for local␣
↪users)
    'display_name': # The user's display name
    'locked': # Denotes whether the account can be followed without a follow request
    'created_at': # Account creation time
    'following_count': # How many people they follow
    'followers_count': # How many followers they have
    'statuses_count': # How many statuses they have
    'note': # Their bio
    'url': # Their URL; usually 'https://mastodon.social/users/<acct>'
    'avatar': # URL for their avatar, can be animated
```

(continues on next page)

(continued from previous page)

```
    'header': # URL for their header image, can be animated
    'avatar_static': # URL for their avatar, never animated
    'header_static': # URL for their header image, never animated
    'source': # Additional information - only present for user dict returned
              # from account_verify_credentials()
    'moved_to_account': # If set, an account dict of the account this user has
                        # set up as their moved-to address.
    'bot': # Boolean indicating whether this account is automated.
    'fields': # List of up to four dicts with free-form 'name' and 'value' profile␣
↪info.
}

mastodon.account_verify_credentials()["source"]
# Returns the following dictionary:
{
    'privacy': # The users default visibility setting ("private", "unlisted" or
↪"public")
    'sensitive': # Denotes whether user media should be marked sensitive by default
    'note': # Plain text version of the users bio
}
```

5.2 Toot dicts

```
mastodon.toot("Hello from Python")
# Returns the following dictionary:
{
    'id': # Numerical id of this toot
    'uri': # Descriptor for the toot
        # EG 'tag:mastodon.social,2016-11-25:objectId=<id>:objectType=Status'
    'url': # URL of the toot
    'account': # User dict for the account which posted the status
    'in_reply_to_id': # Numerical id of the toot this toot is in response to
    'in_reply_to_account_id': # Numerical id of the account this toot is in response␣
↪to
    'reblog': # Denotes whether the toot is a reblog. If so, set to the original toot␣
↪dict.
    'content': # Content of the toot, as HTML: '<p>Hello from Python</p>'
    'created_at': # Creation time
    'reblogs_count': # Number of reblogs
    'favourites_count': # Number of favourites
    'reblogged': # Denotes whether the logged in user has boosted this toot
    'favourited': # Denotes whether the logged in user has favourited this toot
    'sensitive': # Denotes whether media attachments to the toot are marked sensitive
    'spoiler_text': # Warning text that should be displayed before the toot content
    'visibility': # Toot visibility ('public', 'unlisted', 'private', or 'direct')
    'mentions': # A list of users dicts mentioned in the toot, as Mention dicts
    'media_attachments': # A list of media dicts of attached files
    'emojis': # A list of custom emojis used in the toot, as Emoji dicts
    'tags': # A list of hashtag used in the toot, as Hashtag dicts
    'application': # Application dict for the client used to post the toot (Does not␣
↪federate
                   # and is therefore always None for remote toots, can also be None␣
↪for
                   # local toots for some legacy applications).
```

(continues on next page)

(continued from previous page)

```
    'language': # The language of the toot, if specified by the server,
                # as ISO 639-1 (two-letter) language code.
    'muted': # Boolean denoting whether the user has muted this status by
             # way of conversation muting
    'pinned': # Boolean denoting whether or not the status is currently pinned for the
              # associated account.
    'replies_count': # The number of replies to this status.
    'card': # A preview card for links from the status, if present at time of␣
↪delivery,
            # as card dict.
    'poll': # A poll dict if a poll is attached to this status.
}
```

5.3 Mention dicts

```
{
    'url': # Mentioned users profile URL (potentially remote)
    'username': # Mentioned users user name (not including domain)
    'acct': # Mentioned users account name (including domain)
    'id': # Mentioned users (local) account ID
}
```

5.4 Scheduled toot dicts

```
mastodon.status_post("text", scheduled_at=the_future)
# Returns the following dictionary:
{
    'id': # Scheduled toot ID (note: Not the id of the toot once it gets posted!)
    'scheduled_at': # datetime object describing when the toot is to be posted
    'params': # Parameters for the scheduled toot, specifically
    {
        'text': # Toot text
        'in_reply_to_id': # ID of the toot this one is a reply to
        'media_ids': # IDs of media attached to this toot
        'sensitive': # Whether this toot is sensitive or not
        'visibility': # Visibility of the toot
        'idempotency': # Idempotency key for the scheduled toot
        'scheduled_at': # Present, but generally "None"
        'spoiler_text': # CW text for this toot
        'application_id': # ID of the application that scheduled the toot
        'poll': # Poll parameters, as a poll dict
    },
    'media_attachments': # Array of media dicts for the attachments to the scheduled␣
↪toot
}
```

5.5 Poll dicts

```
# Returns the following dictionary:
mastodon.poll(id)
{
    'id': # The polls ID
    'expires_at': # The time at which the poll is set to expire
    'expired': # Boolean denoting whether you can still vote in this poll
    'multiple': # Boolean indicating whether it is allowed to vote for more than one␣
↪option
    'votes_count': # Total number of votes cast in this poll
    'voted': # Boolean indicating whether the logged-in user has already voted in␣
↪this poll
    'options': # The poll options as a list of dicts, each option with a title and a
               # votes_count field. votes_count can be None if the poll creator has
               # chosen to hide vote totals until the poll expires and it hasn't yet.
    'emojis': # List of emoji dicts for all emoji used in answer strings
}
```

5.6 Conversation dicts

```
mastodon.conversations()[0]
# Returns the following dictionary:
{
    'id': # The ID of this conversation object
    'unread': # Boolean indicating whether this conversation has yet to be
              # read by the user
    'accounts': # List of accounts (other than the logged-in account) that
                # are part of this conversation
    'last_status': # The newest status in this conversation
}
```

5.7 Hashtag dicts

```
{
    'name': # Hashtag name (not including the #)
    'url': # Hashtag URL (can be remote)
    'history': # List of usage history dicts for up to 7 days. Not present in␣
↪statuses.
}
```

5.8 Hashtag usage history dicts

```
{
    'day': # Date of the day this history dict is for
    'uses': # Number of statuses using this hashtag on that day
    'accounts': # Number of accounts using this hashtag in at least one status on␣
↪that day
}
```

5.9 Emoji dicts

```
{
    'shortcode': # Emoji shortcode, without surrounding colons
    'url': # URL for the emoji image, can be animated
    'static_url': # URL for the emoji image, never animated
}
```

5.10 Application dicts

```
{
    'name': # The applications name
    'website': # The applications website
    'vapid_key': # A vapid key that can be used in web applications
}
```

5.11 Relationship dicts

```
mastodon.account_follow(<numerical id>)
# Returns the following dictionary:
{
    'id': # Numerical id (same one as <numerical id>)
    'following': # Boolean denoting whether the logged-in user follows the specified␣
↪user
    'followed_by': # Boolean denoting whether the specified user follows the logged-
↪in user
    'blocking': # Boolean denoting whether the logged-in user has blocked the␣
↪specified user
    'muting': # Boolean denoting whether the logged-in user has muted the specified␣
↪user
    'muting_notifications': # Boolean denoting wheter the logged-in user has muted␣
↪notifications
                            # related to the specified user
    'requested': # Boolean denoting whether the logged-in user has sent the specified
                 # user a follow request
    'domain_blocking': # Boolean denoting whether the logged-in user has blocked the
                       # specified users domain
    'showing_reblogs': # Boolean denoting whether the specified users reblogs show up␣
↪on the
                       # logged-in users Timeline
    'endorsed': # Boolean denoting wheter the specified user is being endorsed /␣
↪featured by the
                # logged-in user
}
```

5.12 Filter dicts

```
mastodon.account_follow(<numerical id>)
# Returns the following dictionary:
{
    'id': # Numerical id of the filter
    'phrase': # Filtered keyword or phrase
    'context': # List of places where the filters are applied ('home', 'notifications
↪', 'public', 'thread')
    'expires_at': # Expiry date for the filter
    'irreversible': # Boolean denoting if this filter is executed server-side
                    # or if it should be ran client-side.
    'whole_word': # Boolean denoting whether this filter can match partial words
}
```

5.13 Notification dicts

```
mastodon.notifications()[0]
# Returns the following dictionary:
{
    'id': # id of the notification
    'type': # "mention", "reblog", "favourite" or "follow"
    'created_at': # The time the notification was created
    'account': # User dict of the user from whom the notification originates
    'status': # In case of "mention", the mentioning status
              # In case of reblog / favourite, the reblogged / favourited status
}
```

5.14 Context dicts

```
mastodon.status_context(<numerical id>)
# Returns the following dictionary:
{
    'ancestors': # A list of toot dicts
    'descendants': # A list of toot dicts
}
```

5.15 List dicts

```
mastodon.list(<numerical id>)
# Returns the following dictionary:
{
    'id': # id of the list
    'title': # title of the list
}
```

5.16 Media dicts

```
mastodon.media_post("image.jpg", "image/jpeg")
# Returns the following dictionary:
{
    'id': # The ID of the attachment.
    'type': # Media type: 'image', 'video', 'gifv' or 'unknown'.
    'url': # The URL for the image in the local cache
    'remote_url': # The remote URL for the media (if the image is from a remote␣
↪instance)
    'preview_url': # The URL for the media preview
    'text_url': # The display text for the media (what shows up in toots)
    'meta': # Dictionary of two image metadata dicts (see below),
            # 'original' and 'small' (preview). Either may be empty.
            # May additionally contain an "fps" field giving a videos frames per␣
↪second (possibly
            # rounded), and a "length" field giving a videos length in a human-
↪readable format.
            # Note that a video may have an image as preview.
            # May also contain a 'focus' dict.
}

# Metadata dicts (image) - all fields are optional:
{
   'width': # Width of the image in pixels
   'height': # Height of the image in pixels
   'aspect': # Aspect ratio of the image as a floating point number
   'size': # Textual representation of the image size in pixels, e.g. '800x600'
}

# Metadata dicts (video, gifv) - all fields are optional:
{
    'width': # Width of the video in pixels
    'heigh': # Height of the video in pixels
    'frame_rate': # Exact frame rate of the video in frames per second.
                  # Can be an integer fraction (i.e. "20/7")
    'duration': # Duration of the video in seconds
    'bitrate': # Average bit-rate of the video in bytes per second
}

# Focus Metadata dict:
{
    'x': Focus point x coordinate (between -1 and 1)
    'y': Focus point x coordinate (between -1 and 1)
}
```

5.17 Card dicts

```
mastodon.status_card(<numerical id>):
# Returns the following dictionary
{
    'url': # The URL of the card.
    'title': # The title of the card.
    'description': # The description of the card.
    'type': # Embed type: 'link', 'photo', 'video', or 'rich'
    'image': # (optional) The image associated with the card.
```

(continues on next page)

(continued from previous page)

```
    # OEmbed data (all optional):
    'author_name': # Name of the embedded contents author
    'author_url': # URL pointing to the embedded contents author
    'description': # Description of the embedded content
    'width': # Width of the embedded object
    'height': # Height of the embedded object
    'html': # HTML string of the embed
    'provider_name': # Name of the provider from which the embed originates
    'provider_url': # URL pointing to the embeds provider
}
```

5.18 Search result dicts

```
mastodon.search("<query>")
# Returns the following dictionary
{
    'accounts': # List of account dicts resulting from the query
    'hashtags': # List of hashtag dicts resulting from the query
    'statuses': # List of toot dicts resulting from the query
}
```

5.19 Instance dicts

```
mastodon.instance()
# Returns the following dictionary
{
    'description': # A brief instance description set by the admin
    'email': # The admin contact e-mail
    'title': # The instances title
    'uri': # The instances URL
    'version': # The instances mastodon version
    'urls': # Additional URLs dict, presently only 'streaming_api' with the
            # stream websocket address.
    'stats: # A dictionary containing three stats, user_count (number of local users),
            # status_count (number of local statuses) and domain_count (number of␣
↪known
            # instance domains other than this one).
    'contact_account': # Account dict of the primary contact for the instance.
    'languages': # Array of ISO 639-1 (two-letter) language codes the instance
                 # has chosen to advertise.
    'registrations': # Boolean indication whether registrations on this instance are␣
↪open
                     # (True) or not (False).
}
```

5.20 Activity dicts

```
mastodon.instance_activity()[0]
# Returns the following dictionary
{
    'week': # Date of the first day of the week the stats were collected for
    'logins': # Number of users that logged in that week
    'registrations': # Number of new users that week
    'statuses': # Number of statuses posted that week
}
```

5.21 Report dicts

```
mastodon.reports()[0]
# Returns the following dictionary
{
    'id': # Numerical id of the report
    'action_taken': # True if a moderator or admin has processed the
                    # report, False otherwise. Note that no indication as to
                    # what action was taken is given and that an admin simply
                    # marking the report as processed and not doing anything else
                    # will set this field to True. Note also that now that there
                    # is no way to get any updated report lists, this will
                    # always be false.
}
```

5.22 Push subscription dicts

```
mastodon.push_subscription()
# Returns the following dictionary
{
    'id': # Numerical id of the push subscription
    'endpoint': # Endpoint URL for the subscription
    'server_key': # Server pubkey used for signature verification
    'alerts': # Subscribed events - dict that may contain keys 'follow',
              # 'favourite', 'reblog' and 'mention', with value True
              # if webpushes have been requested for those events.
}
```

5.23 Push notification dicts

```
mastodon.push_subscription_decrypt_push(...)
# Returns the following dictionary
{
    'access_token': # Access token that can be used to access the API as the
                    # notified user
    'body': # Text body of the notification
    'icon': # URL to an icon for the notification
    'notification_id': # ID that can be passed to notification() to get the full
                       # notification object,
    'notification_type': # 'mention', 'reblog', 'follow' or 'favourite'
```

(continues on next page)

(continued from previous page)

```
    'preferred_locale': # The users preferred locale
    'title': # Title for the notification
}
```

5.24 Preference dicts

```
mastodon.preferences()
# Returns the following dictionary
{
    'posting:default:visibility': # The default visibility setting for the users␣
↪posts,
                                  # as a string
    'posting:default:sensitive': # Boolean indicating whether the users uploads should
                                 # be marked sensitive by default
    'posting:default:language': # The users default post language, if set (None if␣
↪not)
    'reading:expand:media': # How the user wishes to be shown sensitive media. Can be
                            # 'default' (hide if sensitive), 'hide_all' or 'show_all'
    'reading:expand:spoilers': # Boolean indicating whether the user wishes to expand
                               # content warnings by default
}
```

CHAPTER 6

App registration and user authentication

Before you can use the mastodon API, you have to register your application (which gets you a client key and client secret) and then log in (which gets you an access token). These functions allow you to do those things. Additionally, it is also possible to programmatically register a new user.

For convenience, once you have a client id, secret and access token, you can simply pass them to the constructor of the class, too!

Note that while it is perfectly reasonable to log back in whenever your app starts, registering a new application on every startup is not, so don't do that - instead, register an application once, and then persist your client id and secret. A convenient method for this is provided by the functions dealing with registering the app, logging in and the Mastodon classes constructor.

To talk to an instance different from the flagship instance, specify the api_base_url (usually, just the URL of the instance, i.e. https://mastodon.social/ for the flagship instance). If no protocol is specified, Mastodon.py defaults to https.

static `Mastodon.`**`create_app`**(*client_name, scopes=['read', 'write', 'follow', 'push'], redirect_uris=None, website=None, to_file=None, api_base_url='https://mastodon.social', request_timeout=300, session=None*)

Create a new app with given *client_name* and *scopes* (The basic scropse are "read", "write", "follow" and "push" - more granular scopes are available, please refere to Mastodon documentation for which).

Specify *redirect_uris* if you want users to be redirected to a certain page after authenticating in an oauth flow. You can specify multiple URLs by passing a list. Note that if you wish to use OAuth authentication with redirects, the redirect URI must be one of the URLs specified here.

Specify *to_file* to persist your apps info to a file so you can use them in the constructor. Specify *api_base_url* if you want to register an app on an instance different from the flagship one. Specify *website* to give a website for your app.

Specify *session* with a requests.Session for it to be used instead of the deafult.

Presently, app registration is open by default, but this is not guaranteed to be the case for all future mastodon instances or even the flagship instance in the future.

Returns *client_id* and *client_secret*, both as strings.

`Mastodon.__init__`(*client_id=None, client_secret=None, access_token=None, api_base_url='https://mastodon.social', debug_requests=False, ratelimit_method='wait', ratelimit_pacefactor=1.1, request_timeout=300, mastodon_version=None, version_check_mode='created', session=None*)

Create a new API wrapper instance based on the given *client_secret* and *client_id*. If you give a *client_id* and it is not a file, you must also give a secret. If you specify an *access_token* then you don't need to specify a *client_id*. It is allowed to specify neither - in this case, you will be restricted to only using endpoints that do not require authentication.

You can also specify an *access_token*, directly or as a file (as written by *log_in()*).

Mastodon.py can try to respect rate limits in several ways, controlled by *ratelimit_method*. "throw" makes functions throw a *MastodonRatelimitError* when the rate limit is hit. "wait" mode will, once the limit is hit, wait and retry the request as soon as the rate limit resets, until it succeeds. "pace" works like throw, but tries to wait in between calls so that the limit is generally not hit (How hard it tries to not hit the rate limit can be controlled by ratelimit_pacefactor). The default setting is "wait". Note that even in "wait" and "pace" mode, requests can still fail due to network or other problems! Also note that "pace" and "wait" are NOT thread safe.

Specify *api_base_url* if you wish to talk to an instance other than the flagship one. If a file is given as *client_id*, client ID and secret are read from that file.

By default, a timeout of 300 seconds is used for all requests. If you wish to change this, pass the desired timeout (in seconds) as *request_timeout*.

For fine-tuned control over the requests object use *session* with a requests.Session.

The *mastodon_version* parameter can be used to specify the version of Mastodon that Mastodon.py will expect to be installed on the server. The function will throw an error if an unparseable Version is specified. If no version is specified, Mastodon.py will set *mastodon_version* to the detected version.

The version check mode can be set to "created" (the default behaviour), "changed" or "none". If set to "created", Mastodon.py will throw an error if the version of Mastodon it is connected to is too old to have an endpoint. If it is set to "changed", it will throw an error if the endpoints behaviour has changed after the version of Mastodon that is connected has been released. If it is set to "none", version checking is disabled.

`Mastodon.log_in`(*username=None, password=None, code=None, redirect_uri='urn:ietf:wg:oauth:2.0:oob', refresh_token=None, scopes=['read', 'write', 'follow', 'push'], to_file=None*)

Get the access token for a user.

The username is the e-mail used to log in into mastodon.

Can persist access token to file *to_file*, to be used in the constructor.

Handles password and OAuth-based authorization.

Will throw a *MastodonIllegalArgumentError* if the OAuth or the username / password credentials given are incorrect, and *MastodonAPIError* if all of the requested scopes were not granted.

For OAuth2, obtain a code via having your user go to the url returned by *auth_request_url()* and pass it as the code parameter. In this case, make sure to also pass the same redirect_uri parameter as you used when generating the auth request URL.

Returns the access token as a string.

`Mastodon.auth_request_url`(*client_id=None, redirect_uris='urn:ietf:wg:oauth:2.0:oob', scopes=['read', 'write', 'follow', 'push'], force_login=False*)

Returns the url that a client needs to request an oauth grant from the server.

To log in with oauth, send your user to this URL. The user will then log in and get a code which you can pass to log_in.

scopes are as in *log_in()*, redirect_uris is where the user should be redirected to after authentication. Note that redirect_uris must be one of the URLs given during app registration. When using urn:ietf:wg:oauth:2.0:oob, the code is simply displayed, otherwise it is added to the given URL as the "code" request parameter.

Pass force_login if you want the user to always log in even when already logged into web mastodon (i.e. when registering multiple different accounts in an app).

`Mastodon.`**`create_account`**(*username, password, email, agreement=False, locale='en', scopes=['read', 'write', 'follow', 'push'], to_file=None*)

Creates a new user account with the given username, password and email. "agreement" must be set to true (after showing the user the instances user agreement and having them agree to it), "locale" specifies the language for the confirmation e-mail as an ISO 639-1 (two-letter) language code.

Does not require an access token, but does require a client grant.

By default, this method is rate-limited by IP to 5 requests per 30 minutes.

Returns an access token (just like log_in), which it can also persist to to_file, and sets it internally so that the user is now logged in. Note that this token can only be used after the user has confirmed their e-mail.

Added: Mastodon v2.7.0, last changed: Mastodon v2.7.0

CHAPTER 7

Versioning

Mastodon.py will check if a certain endpoint is available before doing API calls. By default, it checks against the version of Mastodon retrieved on init(), or the version you specified. Mastodon.py can be set (in the constructor) to either check if an endpoint is available at all (this is the default) or to check if the endpoint is available and behaves as in the newest Mastodon version (with regards to parameters as well as return values). Version checking can also be disabled altogether. If a version check fails, Mastodon.py throws a *MastodonVersionError*.

With the following functions, you can make Mastodon.py re-check the server version or explicitly determine if a specific minimum Version is available.

`Mastodon.`**`retrieve_mastodon_version`**`()`

Determine installed mastodon version and set major, minor and patch (not including RC info) accordingly.

Returns the version string, possibly including rc info.

`Mastodon.`**`verify_minimum_version`**`(`*version_str*`)`

Update version info from server and verify that at least the specified version is present.

Returns True if version requirement is satisfied, False if not.

CHAPTER 8

Reading data: Instances

These functions allow you to fetch information associated with the current instance.

`Mastodon.instance()`

Retrieve basic information about the instance, including the URI and administrative contact email.

Does not require authentication.

Returns an *instance dict*.

Added: Mastodon v1.1.0, last changed: Mastodon v2.3.0

`Mastodon.instance_activity()`

Retrieve activity stats about the instance. May be disabled by the instance administrator - throws a MastodonNotFoundError in that case.

Activity is returned for 12 weeks going back from the current week.

Returns a list *activity dicts*.

Added: Mastodon v2.1.2, last changed: Mastodon v2.1.2

`Mastodon.instance_peers()`

Retrieve the instances that this instance knows about. May be disabled by the instance administrator - throws a MastodonNotFoundError in that case.

Returns a list of URL strings.

Added: Mastodon v2.1.2, last changed: Mastodon v2.1.2

CHAPTER 9

Reading data: Timelines

This function allows you to access the timelines a logged in user could see, as well as hashtag timelines and the public (federated) and local timelines.

`Mastodon.`**`timeline`**(*timeline='home'*, *max_id=None*, *min_id=None*, *since_id=None*, *limit=None*)

Fetch statuses, most recent ones first. *timeline* can be 'home', 'local', 'public', 'tag/hashtag' or 'list/id'. See the following functions documentation for what those do. Local hashtag timelines are supported via the *timeline_hashtag()* function.

The default timeline is the "home" timeline.

Media only queries are supported via the *timeline_public()* and *timeline_hashtag()* functions.

Returns a list of *toot dicts*.

Added: Mastodon v1.0.0, last changed: Mastodon v2.6.0

`Mastodon.`**`timeline_home`**(*max_id=None*, *min_id=None*, *since_id=None*, *limit=None*)

Fetch the logged-in users home timeline (i.e. followed users and self).

Returns a list of *toot dicts*.

Added: Mastodon v1.0.0, last changed: Mastodon v2.6.0

`Mastodon.`**`timeline_local`**(*max_id=None*, *min_id=None*, *since_id=None*, *limit=None*)

Fetches the local / instance-wide timeline, not including replies.

Returns a list of *toot dicts*.

Added: Mastodon v1.0.0, last changed: Mastodon v2.6.0

`Mastodon.`**`timeline_public`**(*max_id=None*, *min_id=None*, *since_id=None*, *limit=None*, *only_media=False*)

Fetches the public / visible-network timeline, not including replies.

Set *only_media* to True to retrieve only statuses with media attachments.

Returns a list of *toot dicts*.

Added: Mastodon v1.0.0, last changed: Mastodon v2.6.0

`Mastodon.`**`timeline_hashtag`**(*hashtag*, *local=False*, *max_id=None*, *min_id=None*, *since_id=None*, *limit=None*, *only_media=False*)

Fetch a timeline of toots with a given hashtag. The hashtag parameter should not contain the leading #.

Set *local* to True to retrieve only instance-local tagged posts. Set *only_media* to True to retrieve only statuses with media attachments.

Returns a list of *toot dicts*.

Added: Mastodon v1.0.0, last changed: Mastodon v2.6.0

`Mastodon.`**`timeline_list`**(*id*, *max_id=None*, *min_id=None*, *since_id=None*, *limit=None*)

Fetches a timeline containing all the toots by users in a given list.

Returns a list of *toot dicts*.

Added: Mastodon v2.1.0, last changed: Mastodon v2.6.0

`Mastodon.`**`conversations`**(*max_id=None*, *min_id=None*, *since_id=None*, *limit=None*)

Fetches a users conversations.

Returns a list of *conversation dicts*.

Added: Mastodon v2.6.0, last changed: Mastodon v2.6.0

CHAPTER 10

Reading data: Statuses

These functions allow you to get information about single statuses.

`Mastodon.`**`status`**`(`*id*`)`

Fetch information about a single toot.

Does not require authentication for publicly visible statuses.

Returns a *toot dict*.

Added: Mastodon v1.0.0, last changed: Mastodon v2.0.0

`Mastodon.`**`status_context`**`(`*id*`)`

Fetch information about ancestors and descendants of a toot.

Does not require authentication for publicly visible statuses.

Returns a *context dict*.

Added: Mastodon v1.0.0, last changed: Mastodon v1.0.0

`Mastodon.`**`status_reblogged_by`**`(`*id*`)`

Fetch a list of users that have reblogged a status.

Does not require authentication for publicly visible statuses.

Returns a list of *user dicts*.

Added: Mastodon v1.0.0, last changed: Mastodon v2.1.0

`Mastodon.`**`status_favourited_by`**`(`*id*`)`

Fetch a list of users that have favourited a status.

Does not require authentication for publicly visible statuses.

Returns a list of *user dicts*.

Added: Mastodon v1.0.0, last changed: Mastodon v2.1.0

`Mastodon.`**`status_card`**`(`*id*`)`

Fetch a card associated with a status. A card describes an object (such as an external video or link) embedded into a status.

Does not require authentication for publicly visible statuses.

Returns a *card dict*.

Added: Mastodon v1.0.0, last changed: Mastodon v1.0.0

10.1 Reading data: Scheduled statuses

These functions allow you to get information about scheduled statuses.

`Mastodon.`**`scheduled_statuses`**`()`

Fetch a list of scheduled statuses

Returns a list of *scheduled toot dicts*.

Added: Mastodon v2.7.0, last changed: Mastodon v2.7.0

`Mastodon.`**`scheduled_status`**`(`*id*`)`

Fetch information about the scheduled status with the given id.

Returns a *scheduled toot dict*.

Added: Mastodon v2.7.0, last changed: Mastodon v2.7.0

10.2 Reading data: Polls

This function allows you to get and refresh information about polls.

`Mastodon.`**`poll`**`(`*id*`)`

Fetch information about the poll with the given id

Returns a *poll dict*.

Added: Mastodon v2.8.0, last changed: Mastodon v2.8.0

CHAPTER 11

Reading data: Notifications

This function allows you to get information about a users notifications.

`Mastodon.`**`notifications`**(*id=None*, *max_id=None*, *min_id=None*, *since_id=None*, *limit=None*)

Fetch notifications (mentions, favourites, reblogs, follows) for the logged-in user.

Can be passed an *id* to fetch a single notification.

Returns a list of *notification dicts*.

Added: Mastodon v1.0.0, last changed: Mastodon v2.6.0

CHAPTER 12

Reading data: Accounts

These functions allow you to get information about accounts and their relationships.

`Mastodon.account(id)`

Fetch account information by user *id*.

Does not require authentication.

Returns a *user dict*.

Added: Mastodon v1.0.0, last changed: Mastodon v1.0.0

`Mastodon.account_verify_credentials()`

Fetch logged-in user's account information.

Returns a *user dict* (Starting from 2.1.0, with an additional "source" field).

Added: Mastodon v1.0.0, last changed: Mastodon v2.1.0

`Mastodon.account_statuses(id, only_media=False, pinned=False, exclude_replies=False, max_id=None, min_id=None, since_id=None, limit=None)`

Fetch statuses by user *id*. Same options as *timeline()* are permitted. Returned toots are from the perspective of the logged-in user, i.e. all statuses visible to the logged-in user (including DMs) are included.

If *only_media* is set, return only statuses with media attachments. If *pinned* is set, return only statuses that have been pinned. Note that as of Mastodon 2.1.0, this only works properly for instance-local users. If *exclude_replies* is set, filter out all statuses that are replies.

Does not require authentication.

Returns a list of *toot dicts*.

Added: Mastodon v1.0.0, last changed: Mastodon v2.7.0

`Mastodon.account_following(id, max_id=None, min_id=None, since_id=None, limit=None)`

Fetch users the given user is following.

Returns a list of *user dicts*.

Added: Mastodon v1.0.0, last changed: Mastodon v2.6.0

`Mastodon.account_followers(id, max_id=None, min_id=None, since_id=None, limit=None)`

Fetch users the given user is followed by.

Returns a list of *user dicts*.

Added: Mastodon v1.0.0, last changed: Mastodon v2.6.0

`Mastodon.account_relationships(id)`

Fetch relationship (following, followed_by, blocking, follow requested) of the logged in user to a given account. *id* can be a list.

Returns a list of *relationship dicts*.

Added: Mastodon v1.0.0, last changed: Mastodon v1.4.0

`Mastodon.account_search(q, limit=None, following=False)`

Fetch matching accounts. Will lookup an account remotely if the search term is in the username@domain format and not yet in the database. Set *following* to True to limit the search to users the logged-in user follows.

Returns a list of *user dicts*.

Added: Mastodon v1.0.0, last changed: Mastodon v2.3.0

CHAPTER 13

Reading data: Keyword filters

These functions allow you to get information about keyword filters.

`Mastodon.filters()`

Fetch all of the logged-in users filters.

Returns a list of *filter dicts*. Not paginated.

Added: Mastodon v2.4.3, last changed: Mastodon v2.4.3

`Mastodon.filter(id)`

Fetches information about the filter with the specified *id*.

Returns a *filter dict*.

Added: Mastodon v2.4.3, last changed: Mastodon v2.4.3

`Mastodon.filters_apply(objects, filters, context)`

Helper function: Applies a list of filters to a list of either statuses or notifications and returns only those matched by none. This function will apply all filters that match the context provided in *context*, i.e. if you want to apply only notification-relevant filters, specify 'notifications'. Valid contexts are 'home', 'notifications', 'public' and 'thread'.

Added: Mastodon v2.4.3, last changed: Mastodon v2.4.3

CHAPTER 14

Reading data: Follow suggestions

`Mastodon.`**`suggestions`**`()`

Fetch follow suggestions for the logged-in user.

Returns a list of *user dicts*.

Added: Mastodon v2.4.3, last changed: Mastodon v2.4.3

CHAPTER 15

Reading data: Lists

These functions allow you to view information about lists.

`Mastodon.lists()`

Fetch a list of all the Lists by the logged-in user.

Returns a list of *list dicts*.

Added: Mastodon v2.1.0, last changed: Mastodon v2.1.0

`Mastodon.list(id)`

Fetch info about a specific list.

Returns a *list dict*.

Added: Mastodon v2.1.0, last changed: Mastodon v2.1.0

`Mastodon.list_accounts(id, max_id=None, min_id=None, since_id=None, limit=None)`

Get the accounts that are on the given list. A *limit* of 0 can be specified to get all accounts without pagination.

Returns a list of *user dicts*.

Added: Mastodon v2.1.0, last changed: Mastodon v2.6.0

CHAPTER 16

Reading data: Follows

CHAPTER 17

Reading data: Favourites

`Mastodon.`**`favourites`**(*max_id=None*, *min_id=None*, *since_id=None*, *limit=None*)

Fetch the logged-in user's favourited statuses.

Returns a list of *toot dicts*.

Added: Mastodon v1.0.0, last changed: Mastodon v2.6.0

CHAPTER 18

Reading data: Follow requests

`Mastodon.`**`follow_requests`**(*max_id=None*, *min_id=None*, *since_id=None*, *limit=None*)

Fetch the logged-in user's incoming follow requests.

Returns a list of *user dicts*.

Added: Mastodon v1.0.0, last changed: Mastodon v2.6.0

CHAPTER 19

Reading data: Searching

`Mastodon.search(q, resolve=True, result_type=None, account_id=None, offset=None, min_id=None, max_id=None)`

Fetch matching hashtags, accounts and statuses. Will perform webfinger lookups if resolve is True. Full-text search is only enabled if the instance supports it, and is restricted to statuses the logged-in user wrote or was mentioned in.

result_type can be one of "accounts", "hashtags" or "statuses", to only search for that type of object.

Specify *account_id* to only get results from the account with that id.

offset, *min_id* and *max_id* can be used to paginate.

Returns a *search result dict*, with tags as *hashtag dicts*.

Added: Mastodon v2.8.0, last changed: Mastodon v2.8.0

`Mastodon.search_v2(q, resolve=True, result_type=None, account_id=None, offset=None, min_id=None, max_id=None)`

Identical to *search_v1()*, except in that it returns tags as *hashtag dicts*, has more parameters, and resolves by default.

Returns a *search result dict*.

Added: Mastodon v2.8.0, last changed: Mastodon v2.8.0

CHAPTER 20

Reading data: Mutes and blocks

These functions allow you to get information about accounts that are muted or blocked by the logged in user.

`Mastodon.`**`mutes`**(*max_id=None*, *min_id=None*, *since_id=None*, *limit=None*)

Fetch a list of users muted by the logged-in user.

Returns a list of *user dicts*.

Added: Mastodon v1.1.0, last changed: Mastodon v2.6.0

`Mastodon.`**`blocks`**(*max_id=None*, *min_id=None*, *since_id=None*, *limit=None*)

Fetch a list of users blocked by the logged-in user.

Returns a list of *user dicts*.

Added: Mastodon v1.0.0, last changed: Mastodon v2.6.0

CHAPTER 21

Reading data: Reports (REMOVED IN 2.5.0)

`Mastodon.reports()`

Fetch a list of reports made by the logged-in user.

Returns a list of *report dicts*.

Warning: This method has now finally been removed, and will not work on mastodon versions 2.5.0 and above.

Added: Mastodon v1.1.0, last changed: Mastodon v1.1.0

CHAPTER 22

Reading data: Domain blocks

`Mastodon.`**`domain_blocks`**(*max_id=None*, *min_id=None*, *since_id=None*, *limit=None*)

Fetch the logged-in user's blocked domains.

Returns a list of blocked domain URLs (as strings, without protocol specifier).

Added: Mastodon v1.4.0, last changed: Mastodon v2.6.0

CHAPTER 23

Reading data: Emoji

`Mastodon.`**`custom_emojis`**`()`

Fetch the list of custom emoji the instance has installed.

Does not require authentication.

Returns a list of *emoji dicts*.

Added: Mastodon v2.1.0, last changed: Mastodon v2.1.0

CHAPTER 24

Reading data: Apps

`Mastodon.`**`app_verify_credentials`**`()`

Fetch information about the current application.

Returns an *application dict*.

Added: Mastodon v2.0.0, last changed: Mastodon v2.7.2

CHAPTER 25

Reading data: Endorsements

`Mastodon.`**`endorsements`**`()`

Fetch list of users endorsemed by the logged-in user.

Returns a list of *user dicts*.

Added: Mastodon v2.5.0, last changed: Mastodon v2.5.0

CHAPTER 26

Reading data: Preferences

`Mastodon.`**`preferences`**`()`

Fetch the users preferences, which can be used to set some default options. As of 2.8.0, apps can only fetch, not update preferences.

Returns a *preference dict*.

Added: Mastodon v2.8.0, last changed: Mastodon v2.8.0

CHAPTER 27

Writing data: Statuses

These functions allow you to post statuses to Mastodon and to interact with already posted statuses.

`Mastodon.`**`status_post`**(*status*, *in_reply_to_id=None*, *media_ids=None*, *sensitive=False*, *visibility=None*, *spoiler_text=None*, *language=None*, *idempotency_key=None*, *content_type=None*, *scheduled_at=None*, *poll=None*)

Post a status. Can optionally be in reply to another status and contain media.

media_ids should be a list. (If it's not, the function will turn it into one.) It can contain up to four pieces of media (uploaded via *media_post()*). *media_ids* can also be the *media dicts* returned by *media_post()* - they are unpacked automatically.

The *sensitive* boolean decides whether or not media attached to the post should be marked as sensitive, which hides it by default on the Mastodon web front-end.

The visibility parameter is a string value and accepts any of: 'direct' - post will be visible only to mentioned users 'private' - post will be visible only to followers 'unlisted' - post will be public but not appear on the public timeline 'public' - post will be public

If not passed in, visibility defaults to match the current account's default-privacy setting (starting with Mastodon version 1.6) or its locked setting - private if the account is locked, public otherwise (for Mastodon versions lower than 1.6).

The *spoiler_text* parameter is a string to be shown as a warning before the text of the status. If no text is passed in, no warning will be displayed.

Specify *language* to override automatic language detection. The parameter accepts all valid ISO 639-2 language codes.

You can set *idempotency_key* to a value to uniquely identify an attempt at posting a status. Even if you call this function more than once, if you call it with the same *idempotency_key*, only one status will be created.

Pass a datetime as *scheduled_at* to schedule the toot for a specific time (the time must be at least 5 minutes into the future). If this is passed, status_post returns a *scheduled toot dict* instead.

Pass *poll* to attach a poll to the status. An appropriate object can be constructed using *make_poll()*

Specify *content_type* to set the content type of your post on Pleroma. It accepts 'text/plain' (default), 'text/markdown', and 'text/html'. This parameter is not supported on Mastodon servers, but will be safely ignored if set.

Returns a *toot dict* with the new status.

Added: Mastodon v1.0.0, last changed: Mastodon v2.8.0

`Mastodon.status_reply`(*to_status*, *status*, *media_ids=None*, *sensitive=False*, *visibility=None*, *spoiler_text=None*, *language=None*, *idempotency_key=None*, *content_type=None*, *scheduled_at=None*, *poll=None*, *untag=False*)

Helper function - acts like status_post, but prepends the name of all the users that are being replied to to the status text and retains CW and visibility if not explicitly overridden.

Set *untag* to True if you want the reply to only go to the user you are replying to, removing every other mentioned user from the conversation.

Added: Mastodon v1.0.0, last changed: Mastodon v2.8.0

`Mastodon.toot`(*status*)

Synonym for *status_post()* that only takes the status text as input.

Usage in production code is not recommended.

Returns a *toot dict* with the new status.

Added: Mastodon v1.0.0, last changed: Mastodon v2.8.0

`Mastodon.make_poll`(*options*, *expires_in*, *multiple=False*, *hide_totals=False*)

Generate a poll object that can be passed as the *poll* option when posting a status.

options is an array of strings with the poll options (Maximum, by default: 4), expires_in is the time in seconds for which the poll should be open. Set multiple to True to allow people to choose more than one answer. Set hide_totals to True to hide the results of the poll until it has expired.

Added: Mastodon v2.8.0, last changed: Mastodon v2.8.0

`Mastodon.status_reblog`(*id*, *visibility=None*)

Reblog / boost a status.

The visibility parameter functions the same as in *status_post()* and allows you to reduce the visibility of a reblogged status.

Returns a *toot dict* with a new status that wraps around the reblogged one.

Added: Mastodon v1.0.0, last changed: Mastodon v2.0.0

`Mastodon.status_unreblog`(*id*)

Un-reblog a status.

Returns a *toot dict* with the status that used to be reblogged.

Added: Mastodon v1.0.0, last changed: Mastodon v2.0.0

`Mastodon.status_favourite`(*id*)

Favourite a status.

Returns a *toot dict* with the favourited status.

Added: Mastodon v1.0.0, last changed: Mastodon v2.0.0

`Mastodon.status_unfavourite`(*id*)

Un-favourite a status.

Returns a *toot dict* with the un-favourited status.

Added: Mastodon v1.0.0, last changed: Mastodon v2.0.0

`Mastodon.status_mute(id)`
Mute notifications for a status.

Returns a *toot dict* with the now muted status

Added: Mastodon v1.4.0, last changed: Mastodon v2.0.0

`Mastodon.status_unmute(id)`
Unmute notifications for a status.

Returns a *toot dict* with the status that used to be muted.

Added: Mastodon v1.4.0, last changed: Mastodon v2.0.0

`Mastodon.status_pin(id)`
Pin a status for the logged-in user.

Returns a *toot dict* with the now pinned status

Added: Mastodon v2.1.0, last changed: Mastodon v2.1.0

`Mastodon.status_unpin(id)`
Unpin a pinned status for the logged-in user.

Returns a *toot dict* with the status that used to be pinned.

Added: Mastodon v2.1.0, last changed: Mastodon v2.1.0

`Mastodon.status_delete(id)`
Delete a status

Added: Mastodon v1.0.0, last changed: Mastodon v1.0.0

27.1 Writing data: Scheduled statuses

Mastodon allows you to schedule statuses (using *status_post()*. The functions in this section allow you to update or delete thusly scheduled statuses.

`Mastodon.scheduled_status_update(id, scheduled_at)`
Update the scheduled time of a scheduled status.

New time must be at least 5 minutes into the future.

Returns a *scheduled toot dict*

Added: Mastodon v2.7.0, last changed: Mastodon v2.7.0

`Mastodon.scheduled_status_delete(id)`
Deletes a scheduled status.

Added: Mastodon v2.7.0, last changed: Mastodon v2.7.0

27.2 Writing data: Polls

This function allows you to vote in polls.

`Mastodon.poll_vote(id, choices)`
Vote in the given poll.

choices is the index of the choice you wish to register a vote for (i.e. its index in the corresponding polls *options* field. In case of a poll that allows selection of more than one option, a list of indices can be passed.

You can only submit choices for any given poll once in case of single-option polls, or only once per option in case of multi-option polls.

Returns the updated *poll dict*

Added: Mastodon v2.8.0, last changed: Mastodon v2.8.0

CHAPTER 28

Writing data: Notifications

These functions allow you to clear all or some notifications.

`Mastodon.`**`notifications_clear`**`()`

Clear out a users notifications

Added: Mastodon v1.0.0, last changed: Mastodon v1.0.0

`Mastodon.`**`notifications_dismiss`**`(`*id*`)`

Deletes a single notification

Added: Mastodon v1.3.0, last changed: Mastodon v1.3.0

CHAPTER 29

Writing data: Conversations

This function allows you to mark conversations read.

`Mastodon.`**`conversations_read`**`(`*id*`)`

Marks a single conversation as read.

Returns the updated *conversation dict*.

WARNING: This method is currently not documented in the official API and might therefore be unstable.

Added: Mastodon v2.6.0, last changed: Mastodon v2.6.0

CHAPTER 30

Writing data: Accounts

These functions allow you to interact with other accounts: To (un)follow and (un)block.

`Mastodon.`**`account_follow`**(*id*, *reblogs=True*)

Follow a user.

Set *reblogs* to False to hide boosts by the followed user.

Returns a *relationship dict* containing the updated relationship to the user.

Added: Mastodon v1.0.0, last changed: Mastodon v2.4.3

`Mastodon.`**`follows`**(*uri*)

Follow a remote user by uri (username@domain).

Returns a *user dict*.

Added: Mastodon v1.0.0, last changed: Mastodon v2.1.0

`Mastodon.`**`account_unfollow`**(*id*)

Unfollow a user.

Returns a *relationship dict* containing the updated relationship to the user.

Added: Mastodon v1.0.0, last changed: Mastodon v1.4.0

`Mastodon.`**`account_block`**(*id*)

Block a user.

Returns a *relationship dict* containing the updated relationship to the user.

Added: Mastodon v1.0.0, last changed: Mastodon v1.4.0

`Mastodon.`**`account_unblock`**(*id*)

Unblock a user.

Returns a *relationship dict* containing the updated relationship to the user.

Added: Mastodon v1.0.0, last changed: Mastodon v1.4.0

`Mastodon.account_mute(id, notifications=True)`

Mute a user.

Set *notifications* to False to receive notifications even though the user is muted from timelines.

Returns a *relationship dict* containing the updated relationship to the user.

Added: Mastodon v1.1.0, last changed: Mastodon v2.4.3

`Mastodon.account_unmute(id)`

Unmute a user.

Returns a *relationship dict* containing the updated relationship to the user.

Added: Mastodon v1.1.0, last changed: Mastodon v1.4.0

`Mastodon.account_pin(id)`

Pin / endorse a user.

Returns a *relationship dict* containing the updated relationship to the user.

Added: Mastodon v2.5.0, last changed: Mastodon v2.5.0

`Mastodon.account_unpin(id)`

Unpin / un-endorse a user.

Returns a *relationship dict* containing the updated relationship to the user.

Added: Mastodon v2.5.0, last changed: Mastodon v2.5.0

`Mastodon.account_update_credentials(display_name=None, note=None, avatar=None, avatar_mime_type=None, header=None, header_mime_type=None, locked=None, fields=None)`

Update the profile for the currently logged-in user.

'note' is the user's bio.

'avatar' and 'header' are images. As with media uploads, it is possible to either pass image data and a mime type, or a filename of an image file, for either.

'locked' specifies whether the user needs to manually approve follow requests.

'fields' can be a list of up to four name-value pairs (specified as tuples) to appear as semi-structured information in the users profile.

Returns the updated *user dict* of the logged-in user.

Added: Mastodon v1.1.1, last changed: Mastodon v2.4.0

CHAPTER 31

Writing data: Keyword filters

These functions allow you to manipulate keyword filters.

`Mastodon.`**`filter_create`**(*phrase*, *context*, *irreversible=False*, *whole_word=True*, *expires_in=None*)

Creates a new keyword filter. *phrase* is the phrase that should be filtered out, *context* specifies from where to filter the keywords. Valid contexts are 'home', 'notifications', 'public' and 'thread'.

Set *irreversible* to True if you want the filter to just delete statuses server side. This works only for the 'home' and 'notifications' contexts.

Set *whole_word* to False if you want to allow filter matches to start or end within a word, not only at word boundaries.

Set *expires_in* to specify for how many seconds the filter should be kept around.

Returns the *filter dict* of the newly created filter.

Added: Mastodon v2.4.3, last changed: Mastodon v2.4.3

`Mastodon.`**`filter_update`**(*id*, *phrase=None*, *context=None*, *irreversible=None*, *whole_word=None*, *expires_in=None*)

Updates the filter with the given *id*. Parameters are the same as in *filter_create()*.

Returns the *filter dict* of the updated filter.

Added: Mastodon v2.4.3, last changed: Mastodon v2.4.3

`Mastodon.`**`filter_delete`**(*id*)

Deletes the filter with the given *id*.

Added: Mastodon v2.4.3, last changed: Mastodon v2.4.3

CHAPTER 32

Writing data: Follow suggestions

`Mastodon.`**`suggestion_delete`**`(`*account_id*`)`

Remove the user with the given *account_id* from the follow suggestions.

Added: Mastodon v2.4.3, last changed: Mastodon v2.4.3

CHAPTER 33

Writing data: Lists

These functions allow you to create, maintain and delete lists.

When creating lists, note that a user can only have a maximum of 50 lists.

`Mastodon.`**`list_create`**(*title*)

Create a new list with the given *title*.

Returns the *list dict* of the created list.

Added: Mastodon v2.1.0, last changed: Mastodon v2.1.0

`Mastodon.`**`list_update`**(*id*, *title*)

Update info about a list, where "info" is really the lists *title*.

Returns the *list dict* of the modified list.

Added: Mastodon v2.1.0, last changed: Mastodon v2.1.0

`Mastodon.`**`list_delete`**(*id*)

Delete a list.

Added: Mastodon v2.1.0, last changed: Mastodon v2.1.0

`Mastodon.`**`list_accounts_add`**(*id*, *account_ids*)

Add the account(s) given in *account_ids* to the list.

Added: Mastodon v2.1.0, last changed: Mastodon v2.1.0

`Mastodon.`**`list_accounts_delete`**(*id*, *account_ids*)

Remove the account(s) given in *account_ids* from the list.

Added: Mastodon v2.1.0, last changed: Mastodon v2.1.0

CHAPTER 34

Writing data: Follow requests

These functions allow you to accept or reject incoming follow requests.

`Mastodon.`**`follow_request_authorize`**`(`*id*`)`

Accept an incoming follow request.

Added: Mastodon v1.0.0, last changed: Mastodon v1.0.0

`Mastodon.`**`follow_request_reject`**`(`*id*`)`

Reject an incoming follow request.

Added: Mastodon v1.0.0, last changed: Mastodon v1.0.0

CHAPTER 35

Writing data: Media

This function allows you to upload media to Mastodon. The returned media IDs (Up to 4 at the same time) can then be used with post_status to attach media to statuses.

`Mastodon.`**`media_post`**(*media_file*, *mime_type=None*, *description=None*, *focus=None*)

Post an image. *media_file* can either be image data or a file name. If image data is passed directly, the mime type has to be specified manually, otherwise, it is determined from the file name. *focus* should be a tuple of floats between -1 and 1, giving the x and y coordinates of the images focus point for cropping (with the origin being the images center).

Throws a *MastodonIllegalArgumentError* if the mime type of the passed data or file can not be determined properly.

Returns a *media dict*. This contains the id that can be used in status_post to attach the media file to a toot.

Added: Mastodon v1.0.0, last changed: Mastodon v2.3.0

`Mastodon.`**`media_update`**(*id*, *description=None*, *focus=None*)

Update the metadata of the media file with the given *id*. *description* and *focus* are as in *media_post()* .

Returns the updated *media dict*.

Added: Mastodon v2.3.0, last changed: Mastodon v2.3.0

CHAPTER 36

Writing data: Reports

`Mastodon.report(account_id, status_ids=None, comment=None, forward=False)`

Report statuses to the instances administrators.

Accepts a list of toot IDs associated with the report, and a comment.

Set forward to True to forward a report of a remote user to that users instance as well as sending it to the instance local administrators.

Returns a *report dict*.

Added: Mastodon v1.1.0, last changed: Mastodon v2.5.0

CHAPTER 37

Writing data: Domain blocks

These functions allow you to block and unblock all statuses from a domain for the logged-in user.

`Mastodon.`**`domain_block`**`(`*domain=None*`)`

Add a block for all statuses originating from the specified domain for the logged-in user.

Added: Mastodon v1.4.0, last changed: Mastodon v1.4.0

`Mastodon.`**`domain_unblock`**`(`*domain=None*`)`

Remove a domain block for the logged-in user.

Added: Mastodon v1.4.0, last changed: Mastodon v1.4.0

CHAPTER 38

Pagination

These functions allow for convenient retrieval of paginated data.

`Mastodon.`**`fetch_next`**(*previous_page*)

Fetches the next page of results of a paginated request. Pass in the previous page in its entirety, or the pagination information dict returned as a part of that pages last status ('_pagination_next').

Returns the next page or None if no further data is available.

`Mastodon.`**`fetch_previous`**(*next_page*)

Fetches the previous page of results of a paginated request. Pass in the previous page in its entirety, or the pagination information dict returned as a part of that pages first status ('_pagination_prev').

Returns the previous page or None if no further data is available.

`Mastodon.`**`fetch_remaining`**(*first_page*)

Fetches all the remaining pages of a paginated request starting from a first page and returns the entire set of results (including the first page that was passed in) as a big list.

Be careful, as this might generate a lot of requests, depending on what you are fetching, and might cause you to run into rate limits very quickly.

CHAPTER 39

Streaming

These functions allow access to the streaming API.

If *async* is False, these methods block forever (or until an error is encountered).

If *async* is True, the listener will listen on another thread and these methods will return a handle corresponding to the open connection. If, in addition, *async_reconnect* is True, the thread will attempt to reconnect to the streaming API if any errors are encountered, waiting *async_reconnect_wait_sec* seconds between reconnection attempts. Note that no effort is made to "catch up" - events created while the connection is broken will not be received. If you need to make sure to get absolutely all notifications / deletes / toots, you will have to do that manually, e.g. using the *on_abort* handler to fill in events since the last received one and then reconnecting.

The connection may be closed at any time by calling the handles close() method. The current status of the handler thread can be checked with the handles is_alive() function, and the streaming status can be checked by calling is_receiving().

The streaming functions take instances of *StreamListener* as the *listener* parameter. A *CallbackStreamListener* class that allows you to specify function callbacks directly is included for convenience.

When in not-async mode or async mode without async_reconnect, the stream functions may raise various exceptions: *MastodonMalformedEventError* if a received event cannot be parsed and *MastodonNetworkError* if any connection problems occur.

`Mastodon.`**`stream_user`**(*listener*, *run_async=False*, *timeout=300*, *reconnect_async=False*, *reconnect_async_wait_sec=5*)

Streams events that are relevant to the authorized user, i.e. home timeline and notifications.

Added: Mastodon v1.1.0, last changed: Mastodon v1.4.2

`Mastodon.`**`stream_public`**(*listener*, *run_async=False*, *timeout=300*, *reconnect_async=False*, *reconnect_async_wait_sec=5*)

Streams public events.

Added: Mastodon v1.1.0, last changed: Mastodon v1.4.2

`Mastodon.`**`stream_local`**(*listener*, *run_async=False*, *timeout=300*, *reconnect_async=False*, *reconnect_async_wait_sec=5*)

Streams local public events.

Added: Mastodon v1.1.0, last changed: Mastodon v1.4.2

`Mastodon.stream_hashtag(tag, listener, run_async=False, timeout=300, reconnect_async=False, reconnect_async_wait_sec=5)`

Stream for all public statuses for the hashtag 'tag' seen by the connected instance.

Added: Mastodon v1.1.0, last changed: Mastodon v1.4.2

`Mastodon.stream_list(id, listener, run_async=False, timeout=300, reconnect_async=False, reconnect_async_wait_sec=5)`

Stream events for the current user, restricted to accounts on the given list.

Added: Mastodon v2.1.0, last changed: Mastodon v2.1.0

39.1 StreamListener

class `mastodon.StreamListener`

Callbacks for the streaming API. Create a subclass, override the on_xxx methods for the kinds of events you're interested in, then pass an instance of your subclass to Mastodon.user_stream(), Mastodon.public_stream(), or Mastodon.hashtag_stream().

`StreamListener.on_update(status)`

A new status has appeared! 'status' is the parsed JSON dictionary describing the status.

`StreamListener.on_notification(notification)`

A new notification. 'notification' is the parsed JSON dictionary describing the notification.

`StreamListener.on_delete(status_id)`

A status has been deleted. status_id is the status' integer ID.

`StreamListener.on_conversation(conversation)`

A direct message (in the direct stream) has been received. conversation contains the resulting conversation dict.

`StreamListener.on_abort(err)`

There was a connection error, read timeout or other error fatal to the streaming connection. The exception object about to be raised is passed to this function for reference.

Note that the exception will be raised properly once you return from this function, so if you are using this handler to reconnect, either never return or start a thread and then catch and ignore the exception.

`StreamListener.handle_heartbeat()`

The server has sent us a keep-alive message. This callback may be useful to carry out periodic housekeeping tasks, or just to confirm that the connection is still open.

39.2 CallbackStreamListener

class `mastodon.CallbackStreamListener(update_handler=None, local_update_handler=None, delete_handler=None, notification_handler=None, conversation_handler=None)`

Simple callback stream handler class. Can optionally additionally send local update events to a separate handler.

CHAPTER 40

Push subscriptions

These functions allow you to manage webpush subscriptions and to decrypt received pushes. Note that the intended setup is not mastodon pushing directly to a users client - the push endpoint should usually be a relay server that then takes care of delivering the (encrypted) push to the end user via some mechanism, where it can then be decrypted and displayed.

Mastodon allows an application to have one webpush subscription per user at a time.

`Mastodon.`**`push_subscription`**`()`

Fetch the current push subscription the logged-in user has for this app.

Returns a *push subscription dict*.

Added: Mastodon v2.4.0, last changed: Mastodon v2.4.0

`Mastodon.`**`push_subscription_set`**`(`*endpoint*, *encrypt_params*, *follow_events=None*, *favourite_events=None*, *reblog_events=None*, *mention_events=None*`)`

Sets up or modifies the push subscription the logged-in user has for this app.

endpoint is the endpoint URL mastodon should call for pushes. Note that mastodon requires https for this URL. *encrypt_params* is a dict with key parameters that allow the server to encrypt data for you: A public key *pubkey* and a shared secret *auth*. You can generate this as well as the corresponding private key using the *push_subscription_generate_keys()* function.

The rest of the parameters controls what kind of events you wish to subscribe to.

Returns a *push subscription dict*.

Added: Mastodon v2.4.0, last changed: Mastodon v2.4.0

`Mastodon.`**`push_subscription_update`**`(`*follow_events=None*, *favourite_events=None*, *reblog_events=None*, *mention_events=None*`)`

Modifies what kind of events the app wishes to subscribe to.

Returns the updated *push subscription dict*.

Added: Mastodon v2.4.0, last changed: Mastodon v2.4.0

`Mastodon.push_subscription_generate_keys()`

Generates a private key, public key and shared secret for use in webpush subscriptions.

Returns two dicts: One with the private key and shared secret and another with the public key and shared secret.

`Mastodon.push_subscription_decrypt_push(data, decrypt_params, encryption_header, crypto_key_header)`

Decrypts *data* received in a webpush request. Requires the private key dict from *push_subscription_generate_keys()* (*decrypt_params*) as well as the Encryption and server Crypto-Key headers from the received webpush

Returns the decoded webpush as a *push notification dict*.

CHAPTER 41

Acknowledgements

Mastodon.py contains work by a large amount of contributors, many of which have put significant work into making it a better library. You can find some information about who helped with which particular feature or fix in the changelog.

Python Module Index

m

mastodon, **??**

Index

Symbols

`__init__()` (*mastodon.Mastodon method*), 21

A

`account()` (*mastodon.Mastodon method*), 35
`account_block()` (*mastodon.Mastodon method*), 73
`account_follow()` (*mastodon.Mastodon method*), 73
`account_followers()` (*mastodon.Mastodon method*), 35
`account_following()` (*mastodon.Mastodon method*), 35
`account_mute()` (*mastodon.Mastodon method*), 73
`account_pin()` (*mastodon.Mastodon method*), 74
`account_relationships()` (*mastodon.Mastodon method*), 36
`account_search()` (*mastodon.Mastodon method*), 36
`account_statuses()` (*mastodon.Mastodon method*), 35
`account_unblock()` (*mastodon.Mastodon method*), 73
`account_unfollow()` (*mastodon.Mastodon method*), 73
`account_unmute()` (*mastodon.Mastodon method*), 74
`account_unpin()` (*mastodon.Mastodon method*), 74
`account_update_credentials()` (*mastodon.Mastodon method*), 74
`account_verify_credentials()` (*mastodon.Mastodon method*), 35
`app_verify_credentials()` (*mastodon.Mastodon method*), 59
`auth_request_url()` (*mastodon.Mastodon method*), 22

B

`blocks()` (*mastodon.Mastodon method*), 51

C

`CallbackStreamListener` (*class in mastodon*), 92
`conversations()` (*mastodon.Mastodon method*), 30
`conversations_read()` (*mastodon.Mastodon method*), 71
`create_account()` (*mastodon.Mastodon method*), 23
`create_app()` (*mastodon.Mastodon static method*), 21
`custom_emojis()` (*mastodon.Mastodon method*), 57

D

`domain_block()` (*mastodon.Mastodon method*), 87
`domain_blocks()` (*mastodon.Mastodon method*), 55
`domain_unblock()` (*mastodon.Mastodon method*), 87

E

`endorsements()` (*mastodon.Mastodon method*), 61

F

`favourites()` (*mastodon.Mastodon method*), 45
`fetch_next()` (*mastodon.Mastodon method*), 89
`fetch_previous()` (*mastodon.Mastodon method*), 89
`fetch_remaining()` (*mastodon.Mastodon method*), 89
`filter()` (*mastodon.Mastodon method*), 37
`filter_create()` (*mastodon.Mastodon method*), 75
`filter_delete()` (*mastodon.Mastodon method*), 75
`filter_update()` (*mastodon.Mastodon method*), 75
`filters()` (*mastodon.Mastodon method*), 37
`filters_apply()` (*mastodon.Mastodon method*), 37
`follow_request_authorize()` (*mastodon.Mastodon method*), 81
`follow_request_reject()` (*mastodon.Mastodon method*), 81
`follow_requests()` (*mastodon.Mastodon method*), 47

follows() (*mastodon.Mastodon method*), 73

H

handle_heartbeat() (*mastodon.StreamListener method*), 92

I

instance() (*mastodon.Mastodon method*), 27
instance_activity() (*mastodon.Mastodon method*), 27
instance_peers() (*mastodon.Mastodon method*), 27

L

list() (*mastodon.Mastodon method*), 41
list_accounts() (*mastodon.Mastodon method*), 41
list_accounts_add() (*mastodon.Mastodon method*), 79
list_accounts_delete() (*mastodon.Mastodon method*), 79
list_create() (*mastodon.Mastodon method*), 79
list_delete() (*mastodon.Mastodon method*), 79
list_update() (*mastodon.Mastodon method*), 79
lists() (*mastodon.Mastodon method*), 41
log_in() (*mastodon.Mastodon method*), 22

M

make_poll() (*mastodon.Mastodon method*), 66
mastodon (*module*), 1
media_post() (*mastodon.Mastodon method*), 83
media_update() (*mastodon.Mastodon method*), 83
mutes() (*mastodon.Mastodon method*), 51

N

notifications() (*mastodon.Mastodon method*), 33
notifications_clear() (*mastodon.Mastodon method*), 69
notifications_dismiss() (*mastodon.Mastodon method*), 69

O

on_abort() (*mastodon.StreamListener method*), 92
on_conversation() (*mastodon.StreamListener method*), 92
on_delete() (*mastodon.StreamListener method*), 92
on_notification() (*mastodon.StreamListener method*), 92
on_update() (*mastodon.StreamListener method*), 92

P

poll() (*mastodon.Mastodon method*), 32
poll_vote() (*mastodon.Mastodon method*), 67
preferences() (*mastodon.Mastodon method*), 63
push_subscription() (*mastodon.Mastodon method*), 93
push_subscription_decrypt_push() (*mastodon.Mastodon method*), 94
push_subscription_generate_keys() (*mastodon.Mastodon method*), 93
push_subscription_set() (*mastodon.Mastodon method*), 93
push_subscription_update() (*mastodon.Mastodon method*), 93

R

ratelimit_lastcall (*mastodon.Mastodon attribute*), 3
ratelimit_limit (*mastodon.Mastodon attribute*), 3
ratelimit_remaining (*mastodon.Mastodon attribute*), 3
ratelimit_reset (*mastodon.Mastodon attribute*), 3
report() (*mastodon.Mastodon method*), 85
reports() (*mastodon.Mastodon method*), 53
retrieve_mastodon_version() (*mastodon.Mastodon method*), 25

S

scheduled_status() (*mastodon.Mastodon method*), 32
scheduled_status_delete() (*mastodon.Mastodon method*), 67
scheduled_status_update() (*mastodon.Mastodon method*), 67
scheduled_statuses() (*mastodon.Mastodon method*), 32
search() (*mastodon.Mastodon method*), 49
search_v2() (*mastodon.Mastodon method*), 49
status() (*mastodon.Mastodon method*), 31
status_card() (*mastodon.Mastodon method*), 31
status_context() (*mastodon.Mastodon method*), 31
status_delete() (*mastodon.Mastodon method*), 67
status_favourite() (*mastodon.Mastodon method*), 66
status_favourited_by() (*mastodon.Mastodon method*), 31
status_mute() (*mastodon.Mastodon method*), 67
status_pin() (*mastodon.Mastodon method*), 67
status_post() (*mastodon.Mastodon method*), 65
status_reblog() (*mastodon.Mastodon method*), 66
status_reblogged_by() (*mastodon.Mastodon method*), 31
status_reply() (*mastodon.Mastodon method*), 66
status_unfavourite() (*mastodon.Mastodon method*), 66
status_unmute() (*mastodon.Mastodon method*), 67
status_unpin() (*mastodon.Mastodon method*), 67

status_unreblog() (*mastodon.Mastodon method*), 66
stream_hashtag() (*mastodon.Mastodon method*), 92
stream_list() (*mastodon.Mastodon method*), 92
stream_local() (*mastodon.Mastodon method*), 91
stream_public() (*mastodon.Mastodon method*), 91
stream_user() (*mastodon.Mastodon method*), 91
StreamListener (*class in mastodon*), 92
suggestion_delete() (*mastodon.Mastodon method*), 77
suggestions() (*mastodon.Mastodon method*), 39

T

timeline() (*mastodon.Mastodon method*), 29
timeline_hashtag() (*mastodon.Mastodon method*), 29
timeline_home() (*mastodon.Mastodon method*), 29
timeline_list() (*mastodon.Mastodon method*), 30
timeline_local() (*mastodon.Mastodon method*), 29
timeline_public() (*mastodon.Mastodon method*), 29
toot() (*mastodon.Mastodon method*), 66

V

verify_minimum_version() (*mastodon.Mastodon method*), 25