Peony Documentation

Release 2.1.2

odrling

Dec 26, 2021

Usage

Peony is an asynchronous Twitter API client for Python 3.6+.

If you encounter an error after updating Peony check out the *Breaking changes* section.

CHAPTER 1

Quickstart

1.1 Installation

To install this module simply run:

```
$ pip install 'peony-twitter[all]'
```

This will install all the modules required to make peony run out of the box. You might feel like some of them are not fit for your needs. Check *Advanced installation* for more information about how to install only the modules you will need.

1.2 Authorize your client

You can use *peony.oauth_dance.oauth_dance()* to authorize your client:

```
>>> from peony.oauth_dance import oauth_dance
>>> tokens = oauth_dance(YOUR_CONSUMER_KEY, YOUR_CONSUMER_SECRET)
>>> from peony import PeonyClient
>>> client = PeonyClient(**tokens)
```

This should open a browser to get a pin to authorize your application.

1.3 Getting started

You can easily create a client using *PeonyClient*. Make sure to get your api keys and access tokens from Twitter's application management page and/or to *Authorize your client*

Note: The package name is `peony` and not `peony-twitter`

```
import asyncio

from peony import PeonyClient

loop = asyncio.get_event_loop()

# create the client using your api keys
client = PeonyClient(consumer_key=YOUR_CONSUMER_KEY,
                     consumer_secret=YOUR_CONSUMER_SECRET,
                     access_token=YOUR_ACCESS_TOKEN,
                     access_token_secret=YOUR_ACCESS_TOKEN_SECRET)

async def getting_started():
    # print your twitter username or screen name
    user = await client.user
    print("I am @%s" % user.screen_name)
    # tweet about your sudden love for peony
    await client.api.statuses.update.post(status="I'm using Peony!!")

# run the coroutine
loop.run_until_complete(getting_started())
```

CHAPTER 2

How to access the API

You can easily access any Twitter API endpoint. Just search for the endpoint that you need on Twitter's documentation, then you can make a request to this endpoint as:

```
client.twitter_subdomain.path.to.endpoint.method()
```

So to access GET statuses/home_timeline:

```
client.api.statuses.home_timeline.get()
```

For a more complete example:

```
# NOTE: any reference to a `creds` variable in the documentation
# examples should have this format
creds = dict(consumer_key=YOUR_CONSUMER_KEY,
             consumer_secret=YOUR_CONSUMER_SECRET,
             access_token=YOUR_ACCESS_TOKEN,
             access_token_secret=YOUR_ACCESS_TOKEN_SECRET)

client = PeonyClient(**creds)

# to access api.twitter.com/1.1/statuses/home_timeline.json
# using the GET method with the parameters count and since_id
async def home():
    return await client.api.statuses.home_timeline.get(count=200,
                                                       since_id=20)

# to access api.twitter.com/1.1/statuses/update.json
# using the POST method with the parameter status
async def track():
    return await client.api.statuses.update.post(status="Hello World!")

# would GET subdomain.twitter.com/1.1/path.json if it were
# an API endpoint
async def path():
    return await client.subdomain.path.get()
```

see *Accessing an API using a different API version* to access APIs that do not use the version '1.1'

Note: Some endpoints require the use of characters that cannot be used as attributes such as GET geo/id/:place_id

You can use the brackets instead:

```
id = 20  # any status id would work as long as it exists
client.api.statuses.show[id].get()
```

Note: Arguments with a leading underscore are arguments that are used to change the behavior of peony for the request (e.g. *_headers* to add some additional headers to the request). Arguments without a leading underscore are parameters of the request you send.

2.1 Access the response data of a REST API endpoint

A call to a REST API endpoint should return a `PeonyResponse` object if the request was successful.

```
async def home():
    req = client.api.statuses.home_timeline.get(count=200, since_id=0, tweet_mode=
↪'extended')

    # this is a PeonyResponse object
    response = await req

    # you can iterate over the response object
    for tweet in response:
        # you can access items as you would do in a dictionnary
        user_id = tweet['user']['id']

        # or as you would access an attribute
        username = tweet.user.screen_name

        display_range = tweet.get('display_text_range', None)
        if display_range is not None:
            # get the text from the display range provided in the response
            # if present
            text = tweet.text[display_range[0]:display_range[1]]
        else:
            # just get the text
            text = tweet.text

        print("@{username} ({id}): {text}".format(username=username,
                                                  id=user_id,
                                                  text=text))
```

Note: If `extended_tweet` is present in the response, attributes that are in `tweet.extended_tweet` can be retrieved right from `tweet`:

```
>>> tweet.display_text_range == tweet.extended_tweet.display_text_range
True # if tweet.extended_tweet.display_range exists.
```

Also, getting the `text` attribute of the data should always retrieve the full text of the tweet even when the data is truncated. So, there should be no need to look for a `full_text` attribute.

Note: `tweet.key` and `tweet['key']` are always equivalent, even when the key is an attribute in `extended_tweet` or `text`.

2.2 Access the response data of a Streaming API endpoint

A call to a Streaming API endpoint should return a *`StreamResponse`* object.

```
async def track():
    req = client.stream.statuses.filter.post(track="uwu")

    # req is an asynchronous context
    async with req as stream:
        # stream is an asynchronous iterator
        async for tweet in stream:
            # check that you actually receive a tweet
            if peony.events.tweet(tweet):
                # you can then access items as you would do with a
                # `PeonyResponse` object
                user_id = tweet['user']['id']
                username = tweet.user.screen_name

                msg = "@{username} ({id}): {text}"
                print(msg.format(username=username,
                                 id=user_id,
                                 text=tweet.text))
```

CHAPTER 3

Upload medias

You can easily upload a media with peony:

```
import asyncio
from peony import PeonyClient

# creds being a dictionnary containing your api keys
client = PeonyClient(**creds)

async def upload_media(path):
    media = await client.upload_media(path)
    await client.api.statuses.update.post(status="Wow! Look at this picture!",
                                          media_ids=[media.media_id])

loop = asyncio.get_event_loop()
loop.run_until_complete(upload_media("picture.jpg"))
```

Note: *upload_media()* has a `chunked` parameter that is the "recommended way" to upload a media on Twitter. This allows to upload video and large gifs on Twitter and could help in case of bad connection (only a chunk to reupload). You can set the `chunk_size` parameter to specify the size of a chunk in bytes. You can specify the mime type of the media using the `media_type` parameter and the Twitter media category using the `media_category` of the media (in case that could not be guessed from the mime type by peony)

You can also use an url:

```
async def upload_from_url():
    media = await client.upload_media("http://some.domain.com/pic.jpg")
    # ...
```

or an aiohttp request:

```
async def upload_from_url():
    async with aiohttp.ClientSession() as session:
```

(continues on next page)

(continued from previous page)

```
        async with session.get("http://some.domain.com/pic.jpg") as req:
            media = await client.upload_media("http://some.domain.com/pic.jpg")
```

Note: This is actually what passing the url to upload_media does but creates a new session instead of using an existing session. You shouldn't have any reason to use this.

or a file object:

```
async def upload_from_file():
    with open("picture.jpg") as file:
        media = await client.upload_media(file)
    # ...
```

Note: It is recommended to use `aiofiles` if you are planning to work with files.

or even bytes:

```
async def upload_from_bytes();
    with open("picture.jpg") as file:
        # there might be a better use case
        media = await client.upload_media(file.read())
```

CHAPTER 4

Iterators

Sometimes you need to make several requests to the same API endpoint in order to get all the data you want (e.g. getting more than 200 tweets of an user). Some iterators are included in Peony and usable through the peony.iterators module that deals with the actual iteration, getting all the responses you need.

4.1 Cursor iterators

This is an iterator for endpoints using the *cursor* parameter (e.g. followers/ids.json). The first argument given to the iterator is the coroutine function that will make the request.

```
from peony import PeonyClient

# creds being a dictionnary containing your api keys
client = PeonyClient(**creds)

async def get_followers(user_id, **additional_params):
    request = client.api.followers.ids.get(id=user_id, count=5000,
                                           **additional_params)
    followers_ids = request.iterator.with_cursor()

    followers = []
    async for data in followers_ids:
        followers.extend(data.ids)

    return followers
```

4.2 Max_id iterators

An iterator for endpoints using the *max_id* parameter (e.g. statuses/user_timeline.json):

```
from peony import PeonyClient

client = PeonyClient(**creds)

async def get_tweets(user_id, n_tweets=1600, **additional_params):
    request = client.api.statuses.user_timeline.get(user_id=user_id,
                                                    count=200,
                                                    **additional_params)
    responses = request.iterator.with_max_id()

    user_tweets = []

    async for tweets in responses:
        user_tweets.extend(tweets)

        if len(user_tweets) >= n_tweets:
            user_tweets = user_tweets[:n_tweets]
            break

    return user_tweets
```

4.3 Since_id iterators

An iterator for endpoints using the `since_id` parameter (e.g. GET statuses/home_timeline.json):

```
import asyncio
import html

from peony import PeonyClient

client = peony.PeonyClient(**creds)

async def get_home(since_id=None, **params):
    request = client.api.statuses.home_timeline.get(count=200, **params)
    responses = request.iterator.with_since_id()

    home = []
    async for tweets in responses:
        for tweet in reversed(tweets):
            text = html.unescape(tweet.text)
            print("@{user.screen_name}: {text}".format(user=tweet.user,
                                                       text=text))
            print("-"*10)

        await asyncio.sleep(120)

    return sorted(home, key=lambda tweet: tweet.id)
```

Note: *with_since_id()* has a fill_gaps parameter that will try to find all the tweets that were sent between 2 iterations if it cannot be found in a single request (more than 200 tweets were sent)

```
responses = request.iterator.with_since_id(fill_gaps=True)
```

Note: Both *with_since_id()* and *with_max_id()* have a `force` parameter that can be used in case you need to keep making requests after a request returned no content. Set `force` to `True` if this is the case.

CHAPTER 5

Tasks

The main advantage of an asynchronous client is that it will be able to run multiple tasks... asynchronously. Which is quite interesting here if you want to access several Streaming APIs, or perform some requests periodically while using a Streaming API.

So I tried to make it easier to create such a program.

5.1 Init tasks

By default the *PeonyClient* makes 2 requests after being started:

- account/verify_credentials.json (kept as `self.user`)
- help/twitter_configuration.json (kept as `self.twitter_configuration`)

If you need to have more informations during the initialization of a client you should use the `init_task()` decorator. The methods decorated with this decorator will be started on setup.

```
from peony import PeonyClient, init_tasks

class Client(PeonyClient):

    @init_tasks
    async def get_setting():
        self.settings = await self.api.account.settings.get()

    @init_tasks
    async def get_likes():
        self.likes = await self.api.favorites.list.get(count=200)
```

Note: The attributes user and twitter_configuration are created by the tasks in PeonyClient.init_tasks() which are the respectively the responses from /1.1/account/verify_credentials.json and /1.1/help/configuration.json. So you can access self.user.id in the class and this will give you the id of the authenticated user.

Note: The attribute `twitter_configuration` is used by the method upload_media when it converts your picture

5.2 The `task` decorator

First you will need to create a subclass of PeonyClient and add a *`task()`* decorator to the methods that you want to run.

```
import asyncio
import time

from peony import PeonyClient, task

class AwesomePeonyClient(PeonyClient):

    @staticmethod
    async def wait_awesome_hour():
        """ wait until the next awesome hour """
        await asyncio.sleep(-time.time() % 3600)

    async def send_awesome_tweet(self, status="Peony is awesome!!"):
        """ send an awesome tweet """
        await self.api.statuses.update.post(status=status)

    @task
    async def awesome_loop(self):
        """ send an awesome tweet every hour """
        while True:
            await self.wait_awesome_hour()
            await self.send_awesome_tweet()

    @task
    async def awesome_user(self):
        """ The user using this program must be just as awesome, right? """
        user = await self.api.account.verify_credentials.get()

        print("This is an awesome user", user.screen_name)


def main():
    """ start all the tasks """
    loop = asyncio.get_event_loop()

    # set your api keys here
    awesome_client = AwesomePeonyClient(
        consumer_key=your_consumer_key,
        consumer_secret=your_consumer_secret,
        access_token=your_access_token,
        access_token_secret=your_access_token_secret,
        loop=loop
    )

    awesome_client.run()  # you can also use the run_tasks()
                          # coroutine if you need it
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    main()
```

Note: The *run_tasks()* method can be used instead of *run()* to start the tasks. Just keep in mind that *run()* is a wrapper around *run_tasks()* with some basic features such as handling KeyboardInterrupt and run *close()* when all the tasks are complete.

CHAPTER 6

Streams

Streams can be used in peony using a async iterators (other than that the usage is similar to that of REST API endpoints).

```
from peony import PeonyClient, events
client = peony.PeonyClient(**creds)


async def track():
    stream = client.stream.statuses.filter.post(track="uwu")

    # stream is an asynchronous iterator
    async for tweet in stream:
        # you can then access items as you would do with a
        # `PeonyResponse` object
        if peony.events.tweet(tweet):
            user_id = tweet['user']['id']
            username = tweet.user.screen_name

            msg = "@{username} ({id}): {text}"
            print(msg.format(username=username,
                             id=user_id,
                             text=tweet.text))

if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.run_until_complete(track())
```

CHAPTER 7

Advanced installation

When you install peony using this command:

```
$ pip3 install 'peony-twitter[all]'
```

You install some modules that you may not need. But before deciding to not install these modules you need to know what will change if they are not installed.

7.1 python-magic

You can install it by running:

```
$ pip3 install 'peony-twitter[magic]'
```

`python-magic` is used to find the mimetype of a file. The mimetype of a media has to be given when making a multipart upload. If you don't install this module you will not be able to send large pictures or GIFs from a file path that would not be recognized by the `mimetypes` module (shipped with Python) or from a file object.

If `python-magic` doesn't seem to be used check that libmagic is installed on your system as this module depends on this native library. You can follow the installation instructions of python-magic.

7.2 aiofiles

You can install it by running:

```
$ pip3 install 'peony-twitter[aiofiles]'
```

or directly:

```
$ pip3 install aiofiles
```

When this is installed every file will be opened using aiofiles, thus every read operation will not block the event loop.

Note: magic and aiofiles can be installed using the `media` extra requirement:

```
$ pip3 install 'peony-twitter[media]'
```

7.3 aiohttp

This command will install some optional dependencies of aiohttp:

```
$ pip3 install 'peony-twitter[aiohttp]'
```

or again directly:

```
$ pip3 install cchardet aiodns
```

This will install cchardet and aiodns, which could speed up aiohttp.

7.4 Minimal installation

If you don't need these modules you can run:

```
$ pip3 install peony-twitter
```

You can install these modules later if you change your mind.

CHAPTER 8

Accessing an API using a different API version

There actually two ways:

- create a client with an `api_version` argument
- provide the api version with the subdomain of the api when creating the path to the ressource

8.1 Create a client with a custom api version

```
# creds being a dict with your api_keys
# notice the use of the `suffix` argument to change the default
# extension ('.json')
client = PeonyClient(**creds, api_version='1', suffix='')

# params being the parameters of the request
req = client['ads-api'].accounts[id].reach_estimate.get(**params)
```

8.2 Add a version when creating the request

```
# notice the use of the `_suffix` argument to change the default
# extension for a request

# using a tuple as key
req = client['ads-api', '1'].accounts[id].reach_estimate.get(_suffix='',
                                                             **kwargs)

# using a dict as key
ads = client[dict(api='ads-api', version='1')]
req = ads.accounts[id].reach_estimate.get(**kwargs, _suffix='')
```

You can also add more arguments to the tuple or dictionnary:

```
# with a dictionnary
adsapi = dict(
    api='ads-api',
    version='1',
    suffix='',
    base_url='https://{api}.twitter.com/{version}'
)

req = client[adsapi].accounts[id].reach_estimate.get(**kwargs,)


# with a tuple
ads = client['ads-api', '1', '', 'https://{api}.twitter.com/{version}']
req = ads.accounts[id].reach_estimate.get(**kwargs)
```

Note: Actually I never tried the ads API but this should work fine. And this seemed easier to grasp to me than using a fake api.

CHAPTER 9

Use the Application only authentication

The application only authentication is restricted to some endpoints. See the Twitter documentation page:

```
import peony
from peony import PeonyClient

client = PeonyClient(consumer_key=YOUR_CONSUMER_KEY,
                     consumer_secret=YOUR_CONSUMER_SECRET,
                     bearer_token=YOUR_BEARER_TOKEN,
                     auth=peony.oauth.OAuth2Headers)
```

Note: The `bearer_token` parameter is not necessary.

CHAPTER 10

The loads function used when decoding responses

The responses sent by the Twitter API are commonly JSON data. By default the data is loaded using the *peony.utils.loads* so that each JSON Object is converted to a dict object which allows to access its items as you would access its attribute.

Which means that:

```
response.data
```

returns the same as:

```
response['data']
```

Also when a tweet has a `text` and a `full_text` items it will return the value of the `full_text` item when getting `text`.

```
response.text == response.full_text
```

and in case the text is in the `extended_tweet` item this should also work.

```
response.text == response.extended_tweet.full_text
```

tldr You should not have to care about how to retrieve the full text of a tweet if you're using peony out of the box. It should find it by itself.

10.1 I don't like this, how can I change this

To change this behavior, PeonyClient has a *loads* argument which is the function used when loading the data. So if you don't want to use the syntax above and want use the default Python's dicts, you can pass *json.loads* as argument when you create the client.

```
from peony import PeonyClient
import json

client = PeonyClient(**creds, loads=json.loads)
client.twitter_configuration  # this is a dict object
client.twitter_configuration['photo_sizes']
client.twitter_configuration.photo_sizes  # raises AttributeError
```

You can also use it to change how JSON data is decoded.

```
import peony

def loads(*args, **kwargs):
    """ parse integers as strings """
    return peony.utils.loads(*args, parse_int=str, **kwargs)

client = peony.PeonyClient(**creds, loads=loads)
```

CHAPTER 11

Handle errors for every request

By default *peony.exceptions.RateLimitExceeded* is handled by sleeping until the rate limit resets and the requests are resent on `asyncio.TimeoutError`. If you would handle these exceptions differently or want to handle other exceptions you can use the `error_handler` argument of *PeonyClient*.

```
import asyncio
import async_timeout
import sys

import aiohttp
from peony import PeonyClient, ErrorHandler

# client using application-only authentication
backup_client = PeonyClient(**creds, auth=peony.oauth.OAuth2Headers)

class MyErrorHandler(ErrorHandler):
    """
        try to use backup_client during rate limits
        retry requests three times before giving up
    """

    def __init__(self, request):
        # this will set the request as self.request (REQUIRED)
        super().__init__(request)
        self.tries = 0

    @ErrorHandler.handle(exceptions.RateLimitExceeded)
    async def handle_rate_limits(self):
        """ Retry the request with another client on RateLimitExceeded """
        self.request.client = backup_client
        return ErrorHandler.RETRY

    # You can handle several requests with a single method
    @ErrorHandler.handle(asyncio.TimeoutError, TimeoutError)
    async def handle_timeout_error(self):
```

(continues on next page)

(continued from previous page)

```
        """ retry the request on TimeoutError """
        return ErrorHandler.RETRY

    @ErrorHandler.handle(Exception)
    async def default_handler(self, exception):
        """ retry on other """
        print("exception: %s" % exception)

        self.tries -= 1
        if self.tries > 0:
            return ErrorHandler.RETRY
        else:
            return ErrorHandler.RAISE

    # NOTE: client.api.statuses.home_timeline.get(_tries=5) should try
    # the request 5 times instead of 3
    async def __call__(self, tries=3, **kwargs):
        self.tries = tries
        await return super().__call__(**kwargs)


client = PeonyClient(**creds, error_handler=MyErrorHandler)
```

Your error handler must inherit from *ErrorHandler* For every exception that you want you want to handle you should create a method decorated by *handle()*. This method can return `utils.ErrorHandler.RETRY` if you want another request to be made. By default a function with no return statement will raise the exception, but you can explicitly raise the exception by returning `utils.ErrorHandler.RAISE`.

Note: You can also choose to not use an error handler and disable the default one by setting the `error_handler` argument to `None`. If you want to disable the global error handling for a specific request pass a `_error_handling` argument to this request with a value of `False`.

CHAPTER 12

Use an already created session

If you use aiohttp to make requests on other websites you can pass on the `aiohttp.ClientSession` object to the *`PeonyClient`* on initialisation as the *session* argument.

```
import asyncio

import aiohttp
from peony import PeonyClient

async def client_with_session():
    async with aiohttp.ClientSession() as session:
        # The client will use the session to make requests
        client = PeonyClient(**creds, session=session)
        await client.run_tasks()

if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.run_until_complete(client_with_session())
```

CHAPTER 13

Breaking changes

This page keeps track of the main changes that could cause your current application to stop working when you update Peony.

13.1 Changes in Peony 2.0

13.1.1 Twitter exceptions inherit from HTTP exceptions

The name of the exceptions related to the HTTP status of the response are now prefixed with `HTTP`. Here is a list of those new exceptions:

- *HTTPNotModified*
- *HTTPBadRequest*
- *HTTPUnauthorized*
- *HTTPForbidden*
- *HTTPNotFound*
- *HTTPNotAcceptable*
- *HTTPConflict*
- *HTTPGone*
- *HTTPEnhanceYourCalm*
- *HTTPUnprocessableEntity*
- *HTTPTooManyRequests*
- *HTTPInternalServerError*
- *HTTPBadGateway*
- *HTTPServiceUnavailable*

- *HTTPGatewayTimeout*

The exceptions related to twitter error codes now inherit from those exceptions, this means that the order of execution of your `except` blocks now matter. The "HTTP" exceptions should be handled after the exceptions related to twitter error codes.

This works as expected:

```
except ReadOnlyApplication:
    ...
except HTTPForbidden:
    ...
```

Here, *ReadOnlyApplication* will be caught by the first `except` block instead of the more specific `except ReadOnlyApplication`.

```
except HTTPForbidden:
    ...
except ReadOnlyApplication:
    ...
```

13.1.2 `PeonyClient` doesn't have a `twitter_configuration` attribute anymore

Twitter removed the endpoint used to set this attribute's value, because they never really changed. So you can use constants instead of using the values from this attribute.

Here is an exemple of what this endpoint used to return in case you need it.

13.2 Changes in Peony 1.1

13.2.1 Error Handlers must inherit from `ErrorHandler`

Error handler should now inherit from *ErrorHandler*. This ensures that the exception will correctly be propagated when you make a request. See *Handle errors for every request* for more details on how to create an error handler.

13.2.2 `PeonyClient`'s properties are now awaitables

It wasn't very documented until now, but *PeonyClient* has two properties `user` and `twitter_configuration`. They used to be created during the first request made by the client which led to some weird scenarios where these properties could return `None`.

Now these properties are awaitables, which can make the syntax a bit more complicated to use, but now you will never be left with a `None`.

```
client = PeonyClient(**api_keys)
user = await client.user  # assuming we are in a coroutine
print(user.screen_name)   # "POTUS"
```

13.2.3 Init tasks don't exist anymore

I think nobody used them anyway but just in case anyone did. They were used to create the `user` and `twitter_configuration` properties in *PeonyClient*.

CHAPTER 14

peony package

14.1 Subpackages

14.1.1 peony.commands package

peony.commands.event_handlers module

class peony.commands.event_handlers.**EventHandler**(*func*, *event*, *prefix=None*, *strict=False*)

Bases: *peony.commands.tasks.Task*

classmethod event_handler(*event*, *prefix=None*, ***values*)

class peony.commands.event_handlers.**EventStream**(*client*)

Bases: abc.ABC

start()

stream_request()

class peony.commands.event_handlers.**EventStreams**

Bases: list

check_setup(*client*)

get_task(*client*)

get_tasks(*client*)

setup(*client*)

peony.commands.event_types module

class peony.commands.event_types.**Event**(*func*, *name*)

Bases: object

Represents an event, the handler attribute is an instance of Handler

Parameters

- **func** (*callable*) – a function that returns True when the data received corresponds to an event
- **name** (*str*) – name given to the event

envelope()
returns an *Event* that can be used for site streams

for_user()
returns an *Event* that can be used for site streams

class peony.commands.event_types.**Events**(**args*, ***kwargs*)
Bases: dict

A class to manage event handlers easily

class peony.commands.event_types.**Handler**(*event*)
Bases: object

A decorator, the decorated function is used when the event is detected related to this handler is detected

Parameters **event** (*func*) – a function that returns True when the data received corresponds to an event

with_prefix(*prefix*, *strict=False*)

decorator to handle commands with prefixes

Parameters

- **prefix** (*str*) – the prefix of the command
- **strict** (*bool, optional*) – If set to True the command must be at the beginning of the message. Defaults to False.

Returns a decorator that returns an `EventHandler` instance

Return type function

peony.commands.tasks module

class peony.commands.tasks.**Task**(*func*)
Bases: object

peony.commands.tasks.**task**
alias of *peony.commands.tasks.Task*

peony.commands.utils module

peony.commands.utils.**doc**(*func*)

Find the message shown when someone calls the help command

Parameters **func** (*function*) – the function

Returns The help message for this command

Return type str

peony.commands.utils.**permission_check**(*data*, *command_permissions*, *command=None*, *permissions=None*)

Check the permissions of the user requesting a command

Parameters

- **data** (*dict*) – message data
- **command_permissions** (*dict*) – permissions of the command, contains all the roles as key and users with these permissions as values
- **command** (*function*) – the command that is run
- **permissions** (*tuple or list*) – a list of permissions for the command

Returns True if the user has the right permissions, False otherwise

Return type bool

Module contents

14.2 peony.api module

class peony.api.**APIPath**(*path*, *suffix*, *client*)

Bases: *peony.api.AbstractAPIPath*

Class to make requests to a REST API

Parameters

- **path** (*str*) – Value of _path
- **suffix** (*str*) – suffix to append to the url
- **client** (client.BasePeonyClient) – client used to perform the request

class peony.api.**AbstractAPIPath**(*path*, *suffix*, *client*)

Bases: abc.ABC

The syntactic sugar factory

Every time you get an attribute or an item from an instance of this class this will be appended to its _path until you call a request method (like get or post)

It makes it easy to call any endpoint of the api

The client given as an parameter during the creation of the BaseAPIPath instance can be accessed as the _client attribute of the instance.

> **Warning:** You must create a child class of AbstractAPIPath to perform requests (you have to implement the _request method)

Parameters

- **path** (*str*) – Value of _path

- **suffix** (*str*) – suffix to append to the url
- **client** (client.BasePeonyClient) – client used to perform the request

delete

get

head

option

patch

post

put

url(*suffix=None*)

Build the url using the _path attribute

Parameters **suffix** (*str*) – String to be appended to the url

Returns Path to the endpoint

Return type str

class peony.api.**StreamingAPIPath**(*path*, *suffix*, *client*)

Bases: *peony.api.AbstractAPIPath*

Class to make requests to a Streaming API

Parameters

- **path** (*str*) – Value of _path
- **suffix** (*str*) – suffix to append to the url
- **client** (client.BasePeonyClient) – client used to perform the request

14.3 peony.client module

Peony Clients

BasePeonyClient only handles requests while *PeonyClient* adds some methods that could help when using the Twitter APIs, with a method to upload a media

class peony.client.**BasePeonyClient**(*consumer_key=None*, *consumer_secret=None*, *access_token=None*, *access_token_secret=None*, *bearer_token=None*, *auth=None*, *headers=None*, *streaming_apis=None*, *base_url=None*, *api_version=None*, *suffix='.json'*, *loads=<function loads>*, *error_handler=<class 'peony.utils.DefaultErrorHandler'>*, *session=None*, *proxy=None*, *compression=True*, *user_agent=None*, *encoding=None*, *loop=None*, ***kwargs*)

Bases: object

Access the Twitter API easily

You can create tasks by decorating a function from a child class with `peony.task`

You also attach a `EventStream` to a subclass using the *event_stream()* of the subclass

After creating an instance of the child class you will be able to run all the tasks easily by executing *get_tasks()*

Parameters

- **streaming_apis** (*iterable, optional*) – Iterable containing the streaming APIs subdomains
- **base_url** (*str, optional*) – Format of the url for all the requests
- **api_version** (*str, optional*) – Default API version
- **suffix** (*str, optional*) – Default suffix of API endpoints
- **loads** (*function, optional*) – Function used to load JSON data
- **error_handler** (*function, optional*) – Requests decorator
- **session** (*aiohttp.ClientSession, optional*) – Session to use to make requests
- **proxy** (*str*) – Proxy used with every request
- **compression** (*bool, optional*) – Activate data compression on every requests, defaults to True
- **user_agent** (*str, optional*) – Set a custom user agent header
- **encoding** (*str, optional*) – text encoding of the response from the server
- **loop** (*event loop, optional*) – An event loop, if not specified `asyncio.get_event_loop()` is called

arun()

close()
properly close the client

classmethod event_stream(*event_stream*)
Decorator to attach an event stream to the class

get_tasks()

Get the tasks attached to the instance

Returns List of tasks (`asyncio.Task`)

Return type list

request(*method*, *url*, *future*, *headers=None*, *session=None*, *encoding=None*, ***kwargs*)

Make requests to the REST API

Parameters

- **future** (*asyncio.Future*) – Future used to return the response
- **method** (*str*) – Method to be used by the request
- **url** (*str*) – URL of the resource
- **headers** (`oauth.PeonyHeaders`) – Custom headers (doesn't overwrite *Authorization* headers)

- **session** (*aiohttp.ClientSession, optional*) – Client session used to make the request

Returns Response to the request

Return type data.PeonyResponse

run()
Run the tasks attached to the instance

run_tasks()
Run the tasks attached to the instance

stream_request(*method*, *url*, *headers=None*, *_session=None*, **args*, ***kwargs*)

Make requests to the Streaming API

Parameters

- **method** (*str*) – Method to be used by the request
- **url** (*str*) – URL of the resource
- **headers** (*dict*) – Custom headers (doesn't overwrite *Authorization* headers)
- **_session** (*aiohttp.ClientSession, optional*) – The session to use for this specific request, the session given as argument of `__init__()` is used by default

Returns Stream context for the request

Return type *stream.StreamResponse*

class peony.client.**MetaPeonyClient**
Bases: type

class peony.client.**PeonyClient**(**args*, ***kwargs*)
Bases: *peony.client.BasePeonyClient*

A client with some useful methods for most usages

upload_media(*file_*, *media_type=None*, *media_category=None*, *chunked=True*, *size_limit=None*, ***params*)

upload a media file on twitter

Parameters

- **file** (*str or pathlib.Path or file*) – Path to the file or file object
- **media_type** (*str, optional*) – mime type of the media
- **media_category** (*str, optional*) – Twitter's media category of the media, must be used with `media_type`
- **chunked** (*bool, optional*) – If True, force the use of the chunked upload for the media
- **params** (*dict*) – parameters used when making the request

Returns Response of the request

Return type data_processing.PeonyResponse

14.4 peony.exceptions module

exception peony.exceptions.**AccessNotAllowedByCredentials**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)

Bases: *peony.exceptions.HTTPForbidden*

exception peony.exceptions.**AccountLocked**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)

Bases: *peony.exceptions.HTTPForbidden*

exception peony.exceptions.**AccountSuspended**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)

Bases: *peony.exceptions.HTTPForbidden*

exception peony.exceptions.**ActionNotPermitted**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)

Bases: *peony.exceptions.HTTPForbidden*

exception peony.exceptions.**AlreadyRetweeted**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)

Bases: *peony.exceptions.HTTPForbidden*

exception peony.exceptions.**ApplicationNotAllowedToAccessDirectMessages**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)

Bases: *peony.exceptions.HTTPForbidden*

exception peony.exceptions.**AttachmentURLInvalid**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)

Bases: *peony.exceptions.HTTPBadRequest*

exception peony.exceptions.**AutomatedRequest**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)

Bases: *peony.exceptions.HTTPForbidden*

exception peony.exceptions.**BadAuthentication**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)

Bases: *peony.exceptions.HTTPBadRequest*

exception peony.exceptions.**CallbackURLNotApproved**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)

Bases: *peony.exceptions.HTTPForbidden*

exception peony.exceptions.**CannotMuteYourself**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)

Bases: *peony.exceptions.HTTPForbidden*

exception peony.exceptions.**CannotReportYourselfAsSpam**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)

Bases: *peony.exceptions.HTTPForbidden*

exception peony.exceptions.**CannotSendMessageToNonFollowers**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)

Bases: *peony.exceptions.HTTPForbidden*

exception peony.exceptions.**CannotSendMessageToUser**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)

Bases: *peony.exceptions.HTTPForbidden*

exception peony.exceptions.**CouldNotAuthenticate**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)

Bases: *peony.exceptions.HTTPUnauthorized*

exception peony.exceptions.**DMCharacterLimit**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)

Bases: *peony.exceptions.HTTPForbidden*

exception peony.exceptions.**DesktopApplicationAuth**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)

Bases: *peony.exceptions.HTTPUnauthorized*

exception peony.exceptions.**DoesNotExist**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)

Bases: *peony.exceptions.HTTPNotFound*

exception peony.exceptions.**DuplicatedStatus**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)

Bases: *peony.exceptions.HTTPForbidden*

class peony.exceptions.**ErrorDict**

Bases: dict

A dict to easily add exception associated to a code

code(*code*)

Decorator to associate a code to an exception

exception peony.exceptions.**FollowLimit**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)

Bases: *peony.exceptions.HTTPForbidden*

exception peony.exceptions.**FollowRequestAlreadyChanged**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)

Bases: *peony.exceptions.HTTPForbidden*

exception peony.exceptions.**GIFNotAllowedWithMultipleImages**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)

Bases: *peony.exceptions.HTTPBadRequest*

exception peony.exceptions.**HTTPBadGateway**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)

Bases: *peony.exceptions.PeonyException*

exception peony.exceptions.**HTTPBadRequest**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)
Bases: *peony.exceptions.PeonyException*

exception peony.exceptions.**HTTPConflict**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)
Bases: *peony.exceptions.PeonyException*

exception peony.exceptions.**HTTPEnhanceYourCalm**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)
Bases: *peony.exceptions.PeonyException*

exception peony.exceptions.**HTTPForbidden**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)
Bases: *peony.exceptions.PeonyException*

exception peony.exceptions.**HTTPGatewayTimeout**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)
Bases: *peony.exceptions.PeonyException*

exception peony.exceptions.**HTTPGone**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)
Bases: *peony.exceptions.PeonyException*

exception peony.exceptions.**HTTPInternalServerError**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)
Bases: *peony.exceptions.PeonyException*

exception peony.exceptions.**HTTPNotAcceptable**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)
Bases: *peony.exceptions.PeonyException*

exception peony.exceptions.**HTTPNotFound**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)
Bases: *peony.exceptions.PeonyException*

exception peony.exceptions.**HTTPNotModified**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)
Bases: *peony.exceptions.PeonyException*

exception peony.exceptions.**HTTPServiceUnavailable**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)
Bases: *peony.exceptions.PeonyException*

exception peony.exceptions.**HTTPTooManyRequests**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)
Bases: *peony.exceptions.PeonyException*

exception peony.exceptions.**HTTPUnauthorized**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)
Bases: *peony.exceptions.PeonyException*

exception peony.exceptions.**HTTPUnprocessableEntity**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)
Bases: *peony.exceptions.PeonyException*

exception peony.exceptions.**InternalError**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)
Bases: *peony.exceptions.HTTPInternalServerError*

exception peony.exceptions.**InvalidCoordinates**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)

Bases: *peony.exceptions.HTTPBadRequest*

exception peony.exceptions.**InvalidOrExpiredToken**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)

Bases: *peony.exceptions.HTTPForbidden*

exception peony.exceptions.**InvalidOrSuspendedApplication**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)

Bases: *peony.exceptions.HTTPUnauthorized*

exception peony.exceptions.**InvalidURL**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)

Bases: *peony.exceptions.HTTPBadRequest*

exception peony.exceptions.**MediaIDNotFound**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)

Bases: *peony.exceptions.HTTPBadRequest*

exception peony.exceptions.**MediaIDValidationFailed**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)

Bases: *peony.exceptions.HTTPBadRequest*

exception peony.exceptions.**MediaProcessingError**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)

Bases: *peony.exceptions.PeonyException*

exception peony.exceptions.**MigrateToNewAPI**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)

Bases: *peony.exceptions.HTTPGone*

exception peony.exceptions.**NoLocationAssociatedToIP**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)

Bases: *peony.exceptions.HTTPNotFound*

exception peony.exceptions.**NoUserMatchesQuery**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)

Bases: *peony.exceptions.HTTPNotFound*

exception peony.exceptions.**NotAuthenticated**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)

Bases: *peony.exceptions.HTTPUnauthorized*

exception peony.exceptions.**NotMutingUser**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)

Bases: *peony.exceptions.HTTPForbidden*

exception peony.exceptions.**OverCapacity**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)

Bases: *peony.exceptions.HTTPServiceUnavailable*

exception peony.exceptions.**OwnerMustAllowDMFromAnyone**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)

Bases: *peony.exceptions.HTTPForbidden*

exception peony.exceptions.**ParameterMissing**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)
Bases: *peony.exceptions.HTTPForbidden*

exception peony.exceptions.**PeonyDecodeError**(*exception*, **args*, ***kwargs*)
Bases: *peony.exceptions.PeonyException*

get_message()

exception peony.exceptions.**PeonyException**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)
Bases: Exception

Parent class of all the exceptions of Peony

get_message()

exception peony.exceptions.**PeonyUnavailableMethod**(*message*)
Bases: *peony.exceptions.PeonyException*

exception peony.exceptions.**ProtectedTweet**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)
Bases: *peony.exceptions.HTTPForbidden*

exception peony.exceptions.**RateLimitExceeded**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)
Bases: *peony.exceptions.HTTPTooManyRequests*

Exception raised on rate limit

reset
Time when the limit will be reset

Returns Time when the limit will be reset

Return type int

reset_in
Time in seconds until the limit will be reset

Returns Time in seconds until the limit will be reset

Return type int

exception peony.exceptions.**ReadOnlyApplication**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)
Bases: *peony.exceptions.HTTPForbidden*

exception peony.exceptions.**ReplyToUnavailableTweet**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)
Bases: *peony.exceptions.HTTPForbidden*

exception peony.exceptions.**RetiredEndpoint**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)
Bases: *peony.exceptions.HTTPGone*

exception peony.exceptions.**SSLRequired**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)
Bases: *peony.exceptions.HTTPForbidden*

exception peony.exceptions.**SpamReportLimit**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)
Bases: *peony.exceptions.HTTPForbidden*

exception peony.exceptions.**StatusAlreadyFavorited**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)
Bases: *peony.exceptions.HTTPForbidden*

exception peony.exceptions.**StatusLimit**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)
Bases: *peony.exceptions.HTTPForbidden*

exception peony.exceptions.**StatusNotFound**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)
Bases: *peony.exceptions.HTTPNotFound*

exception peony.exceptions.**StreamLimit**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)
Bases: *peony.exceptions.PeonyException*

exception peony.exceptions.**SubscriptionAlreadyExists**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)
Bases: *peony.exceptions.HTTPConflict*

exception peony.exceptions.**TooManyAttachmentTypes**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)
Bases: *peony.exceptions.HTTPForbidden*

exception peony.exceptions.**TweetIsReplyRestricted**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)
Bases: *peony.exceptions.HTTPForbidden*

exception peony.exceptions.**TweetNoLongerAvailable**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)
Bases: *peony.exceptions.HTTPNotFound*

exception peony.exceptions.**TweetTooLong**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)
Bases: *peony.exceptions.HTTPForbidden*

exception peony.exceptions.**TweetViolatedRules**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)
Bases: *peony.exceptions.TweetNoLongerAvailable*

exception peony.exceptions.**UnableToVerifyCredentials**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)
Bases: *peony.exceptions.HTTPForbidden*

exception peony.exceptions.**UserNotFound**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)
Bases: *peony.exceptions.HTTPNotFound*

exception peony.exceptions.**UserSuspended**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)
Bases: *peony.exceptions.HTTPNotFound*

exception peony.exceptions.**ValueTooLong**(*response=None*, *error=None*, *data=None*, *url=None*, *message=None*)
Bases: *peony.exceptions.HTTPForbidden*

peony.exceptions.**get_error**(*data*)
return the error if there is a corresponding exception

peony.exceptions.**throw**(*response*, *loads=None*, *encoding=None*, ***kwargs*)
Get the response data if possible and raise an exception

14.5 peony.general module

14.6 peony.iterators module

class peony.iterators.**AbstractIterator**(*request*)
Bases: abc.ABC

Asynchronous iterator

Parameters **request** (requests.Request) – Main request

class peony.iterators.**CursorIterator**(*request*)
Bases: *peony.iterators.AbstractIterator*

Iterate using a cursor

Parameters **request** (requests.Request) – Main request

class peony.iterators.**IdIterator**(*request*, *parameter*, *force=False*)
Bases: *peony.iterators.AbstractIterator*

Iterate using ids

It is the parent class of MaxIdIterator and SinceIdIterator

Parameters

- **request** (requests.Request) – Main request
- **parameter** (*str*) – Parameter to change for each request
- **force** (*bool*) – Keep the iterator after empty responses

call_on_response(*response*)
function that prepares for the next request

get_data(*response*)
Get the data from the response

class peony.iterators.**MaxIdIterator**(*request*, *force=False*)
Bases: *peony.iterators.IdIterator*

Iterator for endpoints using max_id

Parameters **request** (requests.Request) – Main request

call_on_response(*data*)
The parameter is set to the id of the tweet at index i - 1

class peony.iterators.**SinceIdIterator**(*request*, *force=True*, *fill_gaps=False*)
Bases: *peony.iterators.IdIterator*

Iterator for endpoints using since_id

Parameters

- **request** (requests.Request) – Main request
- **force** (*bool*) – Keep the iterator after empty responses
- **fill_gaps** (*bool*) – Fill the gaps (if there are more than `count` tweets to get)

call_on_response(*data*)
Try to fill the gaps and strip last tweet from the response if its id is that of the first tweet of the last response

Parameters **data** (*list*) – The response data

set_param(*data*)

peony.iterators.**with_cursor**
alias of *peony.iterators.CursorIterator*

peony.iterators.**with_max_id**
alias of *peony.iterators.MaxIdIterator*

peony.iterators.**with_since_id**
alias of *peony.iterators.SinceIdIterator*

14.7 peony.oauth module

class peony.oauth.**OAuth1Headers**(*consumer_key*, *consumer_secret*, *access_token=None*, *access_token_secret=None*, *compression=True*, *user_agent=None*, *headers=None*)
Bases: *peony.oauth.PeonyHeaders*

Dynamic headers implementing OAuth1

Parameters

- **consumer_key** (*str*) – Your consumer key
- **consumer_secret** (*str*) – Your consumer secret
- **access_token** (*str*) – Your access token
- **access_token_secret** (*str*) – Your access token secret
- ****kwargs** – Other headers

gen_nonce()

gen_signature(*method*, *url*, *params*, *skip_params*, *oauth*)

sign(*method='GET'*, *url=None*, *data=None*, *params=None*, *skip_params=False*, *headers=None*, ***kwargs*)
sign, that is, generate the *Authorization* headers before making a request

class peony.oauth.**OAuth2Headers**(*consumer_key*, *consumer_secret*, *client*, *bearer_token=None*, *compression=True*, *user_agent=None*, *headers=None*)
Bases: *peony.oauth.PeonyHeaders*

Dynamic headers implementing OAuth2

Parameters

- **consumer_key** (*str*) – Your consumer key
- **consumer_secret** (*str*) – Your consumer secret

- **client** (client.BasePeonyClient) – The client to authenticate
- **bearer_token** (str, optional) – Your bearer_token
- ****kwargs** – Other headers

get_basic_authorization()

invalidate_token()

prepare_request(**args*, *oauth2_pass=False*, ***kwargs*)
prepare all the arguments for the request

Parameters **oauth2_pass** (*bool*) – For oauth2 authentication only (don't use it)

Returns Parameters of the request correctly formatted

Return type dict

refresh_token()

sign(*url=None*, **args*, *headers=None*, ***kwargs*)
sign, that is, generate the *Authorization* headers before making a request

token

class peony.oauth.**PeonyHeaders**(*compression=True*, *user_agent=None*, *headers=None*)
Bases: abc.ABC, dict

Dynamic headers for Peony

This is the base class of *OAuth1Headers* and *OAuth2Headers*.

Parameters

- **compression** (*bool, optional*) – If set to True the client will be able to receive compressed responses else it should not happen unless you provide the corresponding header when you make a request. Defaults to True.
- **user_agent** (*str, optional*) – The user agent set in the headers. Defaults to "peony v{version number}"
- **headers** (*dict*) – dict containing custom headers

prepare_request(*method*, *url*, *headers=None*, *skip_params=False*, *proxy=None*, ***kwargs*)
prepare all the arguments for the request

Parameters

- **method** (*str*) – HTTP method used by the request
- **url** (*str*) – The url to request
- **headers** (*dict, optional*) – Additionnal headers
- **proxy** (*str*) – proxy of the request
- **skip_params** (*bool*) – Don't use the parameters to sign the request

Returns Parameters of the request correctly formatted

Return type dict

sign(**args*, *headers=None*, ***kwargs*)
sign, that is, generate the *Authorization* headers before making a request

class peony.oauth.**RawFormData**(*fields: Iterable[Any] = ()*, *quote_fields: bool = True*, *charset: Optional[str] = None*)

Bases: aiohttp.formdata.FormData

peony.oauth.**quote**(*s*)

14.8 peony.oauth_dance module

peony.oauth_dance.**async_oauth2_dance**(*consumer_key*, *consumer_secret*)

oauth2 dance

Parameters

- **consumer_key** (*str*) – Your consumer key
- **consumer_secret** (*str*) – Your consumer secret

Returns Bearer token

Return type str

peony.oauth_dance.**async_oauth_dance**(*consumer_key*, *consumer_secret*, *callback_uri='oob'*)

OAuth dance to get the user's access token

Parameters

- **consumer_key** (*str*) – Your consumer key
- **consumer_secret** (*str*) – Your consumer secret
- **callback_uri** (*str*) – Callback uri, defaults to 'oob'

Returns Access tokens

Return type dict

peony.oauth_dance.**get_access_token**(*consumer_key*, *consumer_secret*, *oauth_token*, *oauth_token_secret*, *oauth_verifier*, ***kwargs*)

get the access token of the user

Parameters

- **consumer_key** (*str*) – Your consumer key
- **consumer_secret** (*str*) – Your consumer secret
- **oauth_token** (*str*) – OAuth token from *get_oauth_token()*
- **oauth_token_secret** (*str*) – OAuth token secret from *get_oauth_token()*
- **oauth_verifier** (*str*) – OAuth verifier from *get_oauth_verifier()*

Returns Access tokens

Return type dict

peony.oauth_dance.**get_oauth_token**(*consumer_key*, *consumer_secret*, *callback_uri='oob'*)

Get a temporary oauth token

Parameters

- **consumer_key** (*str*) – Your consumer key
- **consumer_secret** (*str*) – Your consumer secret
- **callback_uri** (*str, optional*) – Callback uri, defaults to 'oob'

Returns Temporary tokens

Return type dict

peony.oauth_dance.**get_oauth_verifier**(*oauth_token*)
Open authorize page in a browser, print the url if it didn't work

Parameters **oauth_token** (*str*) – The oauth token received in *get_oauth_token()*

Returns The PIN entered by the user

Return type str

peony.oauth_dance.**oauth2_dance**(*consumer_key*, *consumer_secret*, *loop=None*)

oauth2 dance

Parameters

- **consumer_key** (*str*) – Your consumer key
- **consumer_secret** (*str*) – Your consumer secret
- **loop** (*event loop, optional*) – event loop to use

Returns Bearer token

Return type str

peony.oauth_dance.**oauth_dance**(*consumer_key*, *consumer_secret*, *oauth_callback='oob'*, *loop=None*)

OAuth dance to get the user's access token

It calls async_oauth_dance and create event loop of not given

Parameters

- **consumer_key** (*str*) – Your consumer key
- **consumer_secret** (*str*) – Your consumer secret
- **oauth_callback** (*str*) – Callback uri, defaults to 'oob'
- **loop** (*event loop*) – asyncio event loop

Returns Access tokens

Return type dict

peony.oauth_dance.**parse_token**(*response*)
parse the responses containing the tokens

Parameters **response** (*str*) – The response containing the tokens

Returns The parsed tokens

Return type dict

14.9 peony.requests module

class peony.requests.**AbstractRequest**
Bases: abc.ABC

A function that makes a request when called

get_url(*suffix=None*)

static sanitize_params(*method*, ***kwargs*)

Request params can be extracted from the `**kwargs`

Arguments starting with _ will be stripped from it, so they can be used as an argument for the request (eg. "_headers" → "headers" in the kwargs returned by this function while "headers" would be inserted into the parameters of the request)

Parameters

- **method** (*str*) – method to use to make the request
- **kwargs** (*dict*) – Keywords arguments given to the request

Returns New requests parameters, correctly formatted

Return type dict

class peony.requests.**Endpoint**(**request*)
Bases: object

A class representing an endpoint

Parameters

- **api** (api.AbstractAPIPath) – API path of the request
- **method** (*str*) – HTTP method to be used by the request

class peony.requests.**Iterators**(*request*)
Bases: *peony.requests.Endpoint*

Access the iterators from *peony.iterators* right from a request object

class peony.requests.**Request**(*api*, *method*, ***kwargs*)
Bases: _asyncio.Future, *peony.requests.AbstractRequest*

Sends requests to a REST API

Await an instance of Request to get the response of the request. The request is scheduled as soon as the Request object is created.

client

class peony.requests.**RequestFactory**(*api*, *method*)
Bases: *peony.requests.Endpoint*

Requests to REST APIs

Parameters

- **api** (api.AbstractAPIPath) – API path of the request
- **method** (*str*) – HTTP method to be used by the request

class peony.requests.**StreamingRequest**(*api*, *method*)
Bases: *peony.requests.AbstractRequest*

Requests to Streaming APIs

14.10 peony.stream module

class peony.stream.**StreamResponse**(*client*, *session=None*, *loads=<function loads>*, *timeout=10*, ***kwargs*)
Bases: object

Asynchronous iterator for streams

Parameters

- ***args** (*list, optional*) – Positional arguments of the request
- **client** (*client.BasePeonyClient*) – client used to make the request
- **session** (*aiohttp.ClientSession, optional*) – Session used by the request
- **loads** (*function, optional*) – function used to decode the JSON data received
- **timeout** (*int, optional*) – Timeout on connection
- **kwargs** (*dict, optional*) – Keyword parameters of the request

connect()

Create the connection

Returns

Return type self

Raises exception.PeonyException – On a response status in 4xx that are not status 420 or 429 Also on statuses in 1xx or 3xx since this should not be the status received here

init_restart(*error=None*)

Restart the stream on error

Parameters **error** (*bool, optional*) – Whether to print the error or not

restart_stream()
Restart the stream on error

state

14.11 peony.utils module

class peony.utils.**DefaultErrorHandler**(*request*, *tries=3*)
Bases: *peony.utils.ErrorHandler*

The default error_handler

The decorated request will retry infinitely on any handled error The exceptions handled are TimeoutError, asyncio.TimeoutError, exceptions.RateLimitExceeded and exceptions.ServiceUnavailable

handle_client_error = <peony.utils.Handle object>

handle_rate_limits = <peony.utils.Handle object>

handle_service_unavailable = <peony.utils.Handle object>

handle_timeout_error = <peony.utils.Handle object>

class peony.utils.**Entity**(*original: str, entity_type: str, data: Mapping[str, Any]*)
Bases: object

Helper to use Twitter entities

end

start

text
returns text representing the entity

url
returns an url representing the entity

class peony.utils.**ErrorHandler**(*request*)
Bases: object

Basic error handler

This error handler just raises all the exceptions that it receives.

CONTINUE = True

OK = True

RAISE = False

RETRY = True

STOP = False

static handle(**exceptions*)

class peony.utils.**Handle**(*handler*, **exceptions*)
Bases: object

exceptions

handler

class peony.utils.**MetaErrorHandler**
Bases: type

peony.utils.**execute**(*coro*)

run a function or coroutine

Parameters **coro**(*asyncio.coroutine or function*) –

peony.utils.**get_args**(*func*, *skip=0*)

Hackish way to get the arguments of a function

Parameters

- **func** (*callable*) – Function to get the arguments from
- **skip** (*int, optional*) – Arguments to skip, defaults to 0 set it to 1 to skip the `self` argument of a method.

Returns Function's arguments

Return type tuple

peony.utils.**get_category**(*media_type*)

peony.utils.**get_media_metadata**(*data*, *path=None*)

Get all the file's metadata and read any kind of file object

Parameters

- **data** (*bytes*) – first bytes of the file (the mimetype shoudl be guessed from the file headers
- **path** (*str, optional*) – path to the file

Returns

- *str* – The mimetype of the media
- *str* – The category of the media on Twitter

peony.utils.**get_size**(*media*)

Get the size of a file

Parameters **media** (*file object*) – The file object of the media

Returns The size of the file

Return type int

peony.utils.**get_twitter_entities**(*text: str, entities: Mapping[str, Mapping[str, Any]]*) → Iterable[peony.utils.Entity]

Returns twitter entities from an entities dictionnary

Entities are returned is reversed order for ease of use (start and end indexes stay the same if the string is changed in place)

peony.utils.**get_type**(*media*, *path=None*)

Parameters

- **media** (*file object*) – A file object of the image
- **path** (*str, optional*) – The path to the file

Returns

- *str* – The mimetype of the media
- *str* – The category of the media on Twitter

peony.utils.**log_error**(*msg=None*, *exc_info=None*, *logger=None*, ***kwargs*)

log an exception and its traceback on the logger defined

Parameters

- **msg** (*str, optional*) – A message to add to the error
- **exc_info** (*tuple*) – Information about the current exception

- **logger** (*logging.Logger*) – logger to use

peony.utils.**set_debug**()
activates error messages, useful during development

CHAPTER 15

Indices and tables

- genindex
- modindex

Python Module Index

p

Index

F

G

H

I

L

M

N

U

V

W