Telethon Documentation

Release 0.18.2

Lonami

Apr 08, 2018

Installation and Simple Usage

Pure Python 3 Telegram client library. Official Site here. Please follow the links on the index below to navigate from here, or use the menu on the left. Remember to read the *Changelog (Version History)* when you upgrade!

Important: If you're new here, you want to read *Getting Started*. If you're looking for the method reference, you should check *telethon package*.

CHAPTER 1

What is this?

Telegram is a popular messaging application. This library is meant to make it easy for you to write Python programs that can interact with Telegram. Think of it as a wrapper that has already done the heavy job for you, so you can focus on developing an application.

1.1 Getting Started

1.1.1 Simple Installation

```
pip3 install telethon
```

More details: *Installation*

1.1.2 Creating a client

```
from telethon import TelegramClient

# These example values won't work. You must get your own api_id and
# api_hash from https://my.telegram.org, under API Development.
api_id = 12345
api_hash = '0123456789abcdef0123456789abcdef'

client = TelegramClient('session_name', api_id, api_hash)
client.start()
```

More details: *Creating a Client*

1.1.3 Basic Usage

```
# Getting information about yourself
print(client.get_me().stringify())

# Sending a message (you can use 'me' or 'self' to message yourself)
client.send_message('username', 'Hello World from Telethon!')

# Sending a file
client.send_file('username', '/home/myself/Pictures/holidays.jpg')

# Retrieving messages from a chat
from telethon import utils
for message in client.get_message_history('username', limit=10):
    print(utils.get_display_name(message.sender), message.message)

# Listing all the dialogs (conversations you have open)
for dialog in client.get_dialogs(limit=10):
    print(utils.get_display_name(dialog.entity), dialog.draft.message)

# Downloading profile photos (default path is the working directory)
client.download_profile_photo('username')

# Once you have a message with .media (if message.media)
# you can download it using client.download_media():
messages = client.get_message_history('username')
client.download_media(messages[0])
```

More details: *TelegramClient*

1.1.4 Handling Updates

```
from telethon import events

# We need to have some worker running
client.updates.workers = 1

@client.on(events.NewMessage(incoming=True, pattern='(?i)hi'))
def handler(event):
    event.reply('Hello!')

# If you want to handle updates you can't let the script end.
input('Press enter to exit.')
```

More details: *Working with Updates*

You can continue by clicking on the "More details" link below each snippet of code or the "Next" button at the bottom of the page.

1.2 Installation

1.2.1 Automatic Installation

To install Telethon, simply do:

```
pip3 install telethon
```

Needless to say, you must have Python 3 and PyPi installed in your system. See https://python.org and https://pypi.python.org/pypi/pip for more.

If you already have the library installed, upgrade with:

```
pip3 install --upgrade telethon
```

You can also install the library directly from GitHub or a fork:

```
# pip3 install git+https://github.com/LonamiWebs/Telethon.git
or
$ git clone https://github.com/LonamiWebs/Telethon.git
$ cd Telethon/
# pip install -Ue .
```

If you don't have root access, simply pass the `--user` flag to the pip command. If you want to install a specific branch, append `@branch` to the end of the first install command.

By default the library will use a pure Python implementation for encryption, which can be really slow when uploading or downloading files. If you don't mind using a C extension, install cryptg via `pip` or as an extra:

```
pip3 install telethon[cryptg]
```

1.2.2 Manual Installation

1. Install the required `pyaes` (GitHub | PyPi) and `rsa` (GitHub | PyPi) modules:

   ```
   sudo -H pip3 install pyaes rsa
   ```

2. Clone Telethon's GitHub repository: `git clone https://github.com/LonamiWebs/Telethon.git`
3. Enter the cloned repository: `cd Telethon`
4. Run the code generator: `python3 setup.py gen_tl`
5. Done!

To generate the method documentation, `cd docs` and then `python3 generate.py` (if some pages render bad do it twice).

1.2.3 Optional dependencies

If the cryptg is installed, you might notice a speed-up in the download and upload speed, since these are the most cryptographic-heavy part of the library and said module is a C extension. Otherwise, the `pyaes` fallback will be used.

1.3 Creating a Client

Before working with Telegram's API, you need to get your own API ID and hash:

1. Follow this link and login with your phone number.
2. Click under API Development tools.
3. A *Create new application* window will appear. Fill in your application details. There is no need to enter any *URL*, and only the first two fields (*App title* and *Short name*) can currently be changed later.
4. Click on *Create application* at the end. Remember that your **API hash is secret** and Telegram won't let you revoke it. Don't post it anywhere!

Once that's ready, the next step is to create a `TelegramClient`. This class will be your main interface with Telegram's API, and creating one is very simple:

```
from telethon import TelegramClient

# Use your own values here
api_id = 12345
api_hash = '0123456789abcdef0123456789abcdef'

client = TelegramClient('some_name', api_id, api_hash)
```

Note that `'some_name'` will be used to save your session (persistent information such as access key and others) as `'some_name.session'` in your disk. This is by default a database file using Python's `sqlite3`.

Before using the client, you must be connected to Telegram. Doing so is very easy:

```
client.connect() # Must return True, otherwise, try again
```

You may or may not be authorized yet. You must be authorized before you're able to send any request:

```
client.is_user_authorized() # Returns True if you can send requests
```

If you're not authorized, you need to `.sign_in()`:

```
phone_number = '+34600000000'
client.send_code_request(phone_number)
myself = client.sign_in(phone_number, input('Enter code: '))
# If .sign_in raises PhoneNumberUnoccupiedError, use .sign_up instead
# If .sign_in raises SessionPasswordNeeded error, call .sign_in(password=...)
# You can import both exceptions from telethon.errors.
```

Note: If you send the code that Telegram sent you over the app through the app itself, it will expire immediately. You can still send the code through the app by "obfuscating" it (maybe add a magic constant, like `12345`, and then subtract it to get the real code back) or any other technique.

`myself` is your Telegram user. You can view all the information about yourself by doing `print(myself.stringify())`. You're now ready to use the client as you wish! Remember that any object returned by the API has mentioned `.stringify()` method, and printing these might prove useful.

As a full example:

```
client = TelegramClient('anon', api_id, api_hash)
assert client.connect()
if not client.is_user_authorized():
```

```
    client.send_code_request(phone_number)
    me = client.sign_in(phone_number, input('Enter code: '))
```

All of this, however, can be done through a call to `.start()`:

```
client = TelegramClient('anon', api_id, api_hash)
client.start()
```

The code shown is just what `.start()` will be doing behind the scenes (with a few extra checks), so that you know how to sign in case you want to avoid using `input()` (the default) for whatever reason. If no phone or bot token is provided, you will be asked one through `input()`. The method also accepts a `phone=` and `bot_token` parameters.

You can use either, as both will work. Determining which is just a matter of taste, and how much control you need.

Remember that you can get yourself at any time with `client.get_me()`.

Warning: Please note that if you fail to login around 5 times (or change the first parameter of the `TelegramClient`, which is the session name) you will receive a `FloodWaitError` of around 22 hours, so be careful not to mess this up! This shouldn't happen if you're doing things as explained, though.

Note: If you want to use a **proxy**, you have to install PySocks (via pip or manual) and then set the appropriated parameters:

```
import socks
client = TelegramClient('session_id',
    api_id=12345, api_hash='0123456789abcdef0123456789abcdef',
    proxy=(socks.SOCKS5, 'localhost', 4444)
)
```

The `proxy=` argument should be a tuple, a list or a dict, consisting of parameters described here.

1.3.1 Two Factor Authorization (2FA)

If you have Two Factor Authorization (from now on, 2FA) enabled on your account, calling `telethon.TelegramClient.sign_in()` will raise a `SessionPasswordNeededError`. When this happens, just `telethon.TelegramClient.sign_in()` again with a `password=`:

```
import getpass
from telethon.errors import SessionPasswordNeededError

client.sign_in(phone)
try:
    client.sign_in(code=input('Enter code: '))
except SessionPasswordNeededError:
    client.sign_in(password=getpass.getpass())
```

The mentioned `.start()` method will handle this for you as well, but you must set the `password=` parameter beforehand (it won't be asked).

If you don't have 2FA enabled, but you would like to do so through the library, use `client.edit_2fa()`. Be sure to know what you're doing when using this function and you won't run into any problems. Take note that if you want to set only the email/hint and leave the current password unchanged, you need to "redo" the 2fa.

See the examples below:

```
from telethon.errors import EmailUnconfirmedError

# Sets 2FA password for first time:
client.edit_2fa(new_password='supersecurepassword')

# Changes password:
client.edit_2fa(current_password='supersecurepassword',
                new_password='changedmymind')

# Clears current password (i.e. removes 2FA):
client.edit_2fa(current_password='changedmymind', new_password=None)

# Sets new password with recovery email:
try:
    client.edit_2fa(new_password='memes and dreams',
                    email='JohnSmith@example.com')
    # Raises error (you need to check your email to complete 2FA setup.)
except EmailUnconfirmedError:
    # You can put email checking code here if desired.
    pass

# Also take note that unless you remove 2FA or explicitly
# give email parameter again it will keep the last used setting

# Set hint after already setting password:
client.edit_2fa(current_password='memes and dreams',
                new_password='memes and dreams',
                hint='It keeps you alive')
```

1.4 TelegramClient

1.4.1 Introduction

Note: Check the *telethon package* if you're looking for the methods reference instead of this tutorial.

The `TelegramClient` is the central class of the library, the one you will be using most of the time. For this reason, it's important to know what it offers.

Since we're working with Python, one must not forget that we can do `help(client)` or `help(TelegramClient)` at any time for a more detailed description and a list of all the available methods. Calling `help()` from an interactive Python session will always list all the methods for any object, even yours!

Interacting with the Telegram API is done through sending **requests**, this is, any "method" listed on the API. There are a few methods (and growing!) on the `TelegramClient` class that abstract you from the need of manually importing the requests you need.

For instance, retrieving your own user can be done in a single line:

```
myself = client.get_me()
```

Internally, this method has sent a request to Telegram, who replied with the information about your own user, and then the desired information was extracted from their response.

If you want to retrieve any other user, chat or channel (channels are a special subset of chats), you want to retrieve their "entity". This is how the library refers to either of these:

```
# The method will infer that you've passed an username
# It also accepts phone numbers, and will get the user
# from your contact list.
lonami = client.get_entity('lonami')
```

The so called "entities" are another important whole concept on its own, but for now you don't need to worry about it. Simply know that they are a good way to get information about an user, chat or channel.

Many other common methods for quick scripts are also available:

```
# Note that you can use 'me' or 'self' to message yourself
client.send_message('username', 'Hello World from Telethon!')

# .send_message's parse mode defaults to markdown, so you
# can use **bold**, __italics__, [links](https://example.com), `code`,
# and even [mentions](@username)/[mentions](tg://user?id=123456789)
client.send_message('username', '**Using** __markdown__ `too`!')

client.send_file('username', '/home/myself/Pictures/holidays.jpg')

# The utils package has some goodies, like .get_display_name()
from telethon import utils
for message in client.get_message_history('username', limit=10):
    print(utils.get_display_name(message.sender), message.message)

# Dialogs are the conversations you have open
for dialog in client.get_dialogs(limit=10):
    print(utils.get_display_name(dialog.entity), dialog.draft.message)

# Default path is the working directory
client.download_profile_photo('username')

# Call .disconnect() when you're done
client.disconnect()
```

Remember that you can call `.stringify()` to any object Telegram returns to pretty print it. Calling `str(result)` does the same operation, but on a single line.

1.4.2 Available methods

The *reference* lists all the "handy" methods available for you to use in the `TelegramClient` class. These are simply wrappers around the "raw" Telegram API, making it much more manageable and easier to work with.

Please refer to *Accessing the Full API* if these aren't enough, and don't be afraid to read the source code of the InteractiveTelegramClient or even the TelegramClient itself to learn how it works.

See the mentioned *telethon package* to find the available methods.

1.5 Users, Chats and Channels

1.5.1 Introduction

The library widely uses the concept of "entities". An entity will refer to any User, Chat or Channel object that the API may return in response to certain methods, such as GetUsersRequest.

Note: When something "entity-like" is required, it means that you need to provide something that can be turned into an entity. These things include, but are not limited to, usernames, exact titles, IDs, Peer objects, or even entire User, Chat and Channel objects and even phone numbers from people you have in your contacts.

To "encounter" an ID, you would have to "find it" like you would in the normal app. If the peer is in your dialogs, you would need to *client.get_dialogs()*. If the peer is someone in a group, you would similarly *client.get_participants(group)*.

1.5.2 Getting entities

Through the use of the *Session Files*, the library will automatically remember the ID and hash pair, along with some extra information, so you're able to just do this:

```
# Dialogs are the "conversations you have open".
# This method returns a list of Dialog, which
# has the .entity attribute and other information.
dialogs = client.get_dialogs()

# All of these work and do the same.
lonami = client.get_entity('lonami')
lonami = client.get_entity('t.me/lonami')
lonami = client.get_entity('https://telegram.dog/lonami')

# Other kind of entities.
channel = client.get_entity('telegram.me/joinchat/AAAAAEkk2WdoDrB4-Q8-gg')
contact = client.get_entity('+34xxxxxxxxx')
friend  = client.get_entity(friend_id)

# Getting entities through their ID (User, Chat or Channel)
entity = client.get_entity(some_id)

# You can be more explicit about the type for said ID by wrapping
# it inside a Peer instance. This is recommended but not necessary.
from telethon.tl.types import PeerUser, PeerChat, PeerChannel

my_user    = client.get_entity(PeerUser(some_id))
my_chat    = client.get_entity(PeerChat(some_id))
my_channel = client.get_entity(PeerChannel(some_id))
```

All methods in the *TelegramClient* call `.get_input_entity()` prior to sending the requst to save you from the hassle of doing so manually. That way, convenience calls such as `client.send_message('lonami', 'hi!')` become possible.

Every entity the library encounters (in any response to any call) will by default be cached in the `.session` file (an SQLite database), to avoid performing unnecessary API calls. If the entity cannot be found, additonal calls like ResolveUsernameRequest or GetContactsRequest may be made to obtain the required information.

1.5.3 Entities vs. Input Entities

Note: Don't worry if you don't understand this section, just remember some of the details listed here are important. When you're calling a method, don't call `.get_entity()` beforehand, just use the username or phone, or the entity retrieved by other means like `.get_dialogs()`.

On top of the normal types, the API also make use of what they call their `Input*` versions of objects. The input version of an entity (e.g. InputPeerUser, InputChat, etc.) only contains the minimum information that's required from Telegram to be able to identify who you're referring to: a Peer's **ID** and **hash**.

This ID/hash pair is unique per user, so if you use the pair given by another user **or bot** it will **not** work.

To save *even more* bandwidth, the API also makes use of the Peer versions, which just have an ID. This serves to identify them, but peers alone are not enough to use them. You need to know their hash before you can "use them".

As we just mentioned, API calls don't need to know the whole information about the entities, only their ID and hash. For this reason, another method, `.get_input_entity()` is available. This will always use the cache while possible, making zero API calls most of the time. When a request is made, if you provided the full entity, e.g. an User, the library will convert it to the required InputPeer automatically for you.

You should always favour `.get_input_entity()` **over** `.get_entity()` for this reason! Calling the latter will always make an API call to get the most recent information about said entity, but invoking requests don't need this information, just the `InputPeer`. Only use `.get_entity()` if you need to get actual information, like the username, name, title, etc. of the entity.

To further simplify the workflow, since the version `0.16.2` of the library, the raw requests you make to the API are also able to call `.get_input_entity` wherever needed, so you can even do things like:

```
client(SendMessageRequest('username', 'hello'))
```

The library will call the `.resolve()` method of the request, which will resolve `'username'` with the appropriated InputPeer. Don't worry if you don't get this yet, but remember some of the details here are important.

1.6 Working with Updates

The library comes with the `events` module. *Events* are an abstraction over what Telegram calls updates, and are meant to ease simple and common usage when dealing with them, since there are many updates. If you're looking for the method reference, check *telethon.events package*, otherwise, let's dive in!

Note: The library logs by default no output, and any exception that occurs inside your handlers will be "hidden" from you to prevent the thread from terminating (so it can still deliver events). You should enable logging (`import logging; logging.basicConfig(level=logging.ERROR)`) when working with events, at least the error level, to see if this is happening so you can debug the error.

Contents

1.6.1 Getting Started

```
from telethon import TelegramClient, events

client = TelegramClient(..., update_workers=1, spawn_read_thread=False)
client.start()

@client.on(events.NewMessage)
def my_event_handler(event):
    if 'hello' in event.raw_text:
        event.reply('hi!')

client.idle()
```

Not much, but there might be some things unclear. What does this code do?

```
from telethon import TelegramClient, events

client = TelegramClient(..., update_workers=1, spawn_read_thread=False)
client.start()
```

This is normal initialization (of course, pass session name, API ID and hash). Nothing we don't know already.

```
@client.on(events.NewMessage)
```

This Python decorator will attach itself to the `my_event_handler` definition, and basically means that *on* a `NewMessage` *event*, the callback function you're about to define will be called:

```
def my_event_handler(event):
    if 'hello' in event.raw_text:
        event.reply('hi!')
```

If a `NewMessage` event occurs, and `'hello'` is in the text of the message, we `reply` to the event with a `'hi!'` message.

```
client.idle()
```

Finally, this tells the client that we're done with our code, and want to listen for all these events to occur. Of course, you might want to do other things instead idling. For this refer to *Update Modes*.

1.6.2 More on events

The `NewMessage` event has much more than what was shown. You can access the `.sender` of the message through that member, or even see if the message had `.media`, a `.photo` or a `.document` (which you could download with for example `client.download_media(event.photo)`.

If you don't want to `.reply` as a reply, you can use the `.respond()` method instead. Of course, there are more events such as `ChatAction` or `UserUpdate`, and they're all used in the same way. Simply add the `@client.on(events.XYZ)` decorator on the top of your handler and you're done! The event that will be passed always is

of type `XYZ.Event` (for instance, `NewMessage.Event`), except for the `Raw` event which just passes the `Update` object.

Note that `.reply()` and `.respond()` are just wrappers around the `client.send_message()` method which supports the `file=` parameter. This means you can reply with a photo if you do `client.reply(file=photo)`.

You can put the same event on many handlers, and even different events on the same handler. You can also have a handler work on only specific chats, for example:

```
import ast
import random


# Either a single item or a list of them will work for the chats.
# You can also use the IDs, Peers, or even User/Chat/Channel objects.
@client.on(events.NewMessage(chats=('TelethonChat', 'TelethonOffTopic')))
def normal_handler(event):
    if 'roll' in event.raw_text:
        event.reply(str(random.randint(1, 6)))


# Similarly, you can use incoming=True for messages that you receive
@client.on(events.NewMessage(chats='TelethonOffTopic', outgoing=True))
def admin_handler(event):
    if event.raw_text.startswith('eval'):
        expression = event.raw_text.replace('eval', '').strip()
        event.reply(str(ast.literal_eval(expression)))
```

You can pass one or more chats to the `chats` parameter (as a list or tuple), and only events from there will be processed. You can also specify whether you want to handle incoming or outgoing messages (those you receive or those you send). In this example, people can say `'roll'` and you will reply with a random number, while if you say `'eval 4+4'`, you will reply with the solution. Try it!

1.6.3 Events without decorators

If for any reason you can't use the `@client.on` syntax, don't worry. You can call `client.add_event_handler(callback, event)` to achieve the same effect.

Similar to that method, you also have `client.remove_event_handler()` and `client.list_event_handlers()` which do as they names indicate.

The `event` type is optional in all methods and defaults to `events.Raw` for adding, and `None` when removing (so all callbacks would be removed).

1.6.4 Stopping propagation of Updates

There might be cases when an event handler is supposed to be used solitary and it makes no sense to process any other handlers in the chain. For this case, it is possible to raise a `StopPropagation` exception which will cause the propagation of the update through your handlers to stop:

```
from telethon.events import StopPropagation

@client.on(events.NewMessage)
def _(event):
    # ... some conditions
    event.delete()
```

```
    # Other handlers won't have an event to work with
    raise StopPropagation

@client.on(events.NewMessage)
def _(event):
    # Will never be reached, because it is the second handler
    # in the chain.
    pass
```

Remember to check *telethon.events package* if you're looking for the methods reference.

1.7 Accessing the Full API

The `TelegramClient` doesn't offer a method for every single request the Telegram API supports. However, it's very simple to *call* or *invoke* any request. Whenever you need something, don't forget to check the documentation and look for the method you need. There you can go through a sorted list of everything you can do.

Note: The reason to keep both https://lonamiwebs.github.io/Telethon and this documentation alive is that the former allows instant search results as you type, and a "Copy import" button. If you like namespaces, you can also do `from telethon.tl import types, functions`. Both work.

You should also refer to the documentation to see what the objects (constructors) Telegram returns look like. Every constructor inherits from a common type, and that's the reason for this distinction.

Say `client.send_message()` didn't exist, we could use the search to look for "message". There we would find SendMessageRequest, which we can work with.

Every request is a Python class, and has the parameters needed for you to invoke it. You can also call `help(request)` for information on what input parameters it takes. Remember to "Copy import to the clipboard", or your script won't be aware of this class! Now we have:

```
from telethon.tl.functions.messages import SendMessageRequest
```

If you're going to use a lot of these, you may do:

```
from telethon.tl import types, functions
# We now have access to 'functions.messages.SendMessageRequest'
```

We see that this request must take at least two parameters, a `peer` of type InputPeer, and a `message` which is just a Python `string`.

How can we retrieve this InputPeer? We have two options. We manually construct one, for instance:

```
from telethon.tl.types import InputPeerUser

peer = InputPeerUser(user_id, user_hash)
```

Or we call `.get_input_entity()`:

```
peer = client.get_input_entity('someone')
```

When you're going to invoke an API method, most require you to pass an InputUser, InputChat, or so on, this is why using `.get_input_entity()` is more straightforward (and often immediate, if you've seen the user before, know their ID, etc.). If you also need to have information about the whole user, use `.get_entity()` instead:

```
entity = client.get_entity('someone')
```

In the later case, when you use the entity, the library will cast it to its "input" version for you. If you already have the complete user and want to cache its input version so the library doesn't have to do this every time its used, simply call `.get_input_peer`:

```
from telethon import utils
peer = utils.get_input_user(entity)
```

Note: Since `v0.16.2` this is further simplified. The `Request` itself will call `client.get_input_entity()` for you when required, but it's good to remember what's happening.

After this small parenthesis about `.get_entity` versus `.get_input_entity`, we have everything we need. To `.invoke()` our request we do:

```
result = client(SendMessageRequest(peer, 'Hello there!'))
# __call__ is an alias for client.invoke(request). Both will work
```

Message sent! Of course, this is only an example. There are nearly 250 methods available as of layer 73, and you can use every single of them as you wish. Remember to use the right types! To sum up:

```
result = client(SendMessageRequest(
    client.get_input_entity('username'), 'Hello there!'
))
```

This can further be simplified to:

```
result = client(SendMessageRequest('username', 'Hello there!'))
# Or even
result = client(SendMessageRequest(PeerChannel(id), 'Hello there!'))
```

Note: Note that some requests have a "hash" parameter. This is **not** your `api_hash`! It likely isn't your self-user `.access_hash` either.

It's a special hash used by Telegram to only send a difference of new data that you don't already have with that request, so you can leave it to 0, and it should work (which means no hash is known yet).

For those requests having a "limit" parameter, you can often set it to zero to signify "return default amount". This won't work for all of them though, for instance, in "messages.search" it will actually return 0 items.

1.8 Session Files

The first parameter you pass to the constructor of the `TelegramClient` is the `session`, and defaults to be the session name (or full path). That is, if you create a `TelegramClient('anon')` instance and connect, an `anon.session` file will be created on the working directory.

These database files using `sqlite3` contain the required information to talk to the Telegram servers, such as to which IP the client should connect, port, authorization key so that messages can be encrypted, and so on.

These files will by default also save all the input entities that you've seen, so that you can get information about an user or channel by just their ID. Telegram will **not** send their `access_hash` required to retrieve more information about them, if it thinks you have already seem them. For this reason, the library needs to store this information offline.

The library will by default too save all the entities (chats and channels with their name and username, and users with the phone too) in the session file, so that you can quickly access them by username or phone number.

If you're not going to work with updates, or don't need to cache the `access_hash` associated with the entities' ID, you can disable this by setting `client.session.save_entities = False`.

1.8.1 Custom Session Storage

If you don't want to use the default SQLite session storage, you can also use one of the other implementations or implement your own storage.

To use a custom session storage, simply pass the custom session instance to `TelegramClient` instead of the session name.

Telethon contains two implementations of the abstract `Session` class:

- `MemorySession`: stores session data in Python variables.
- `SQLiteSession`, (default): stores sessions in their own SQLite databases.

There are other community-maintained implementations available:

- SQLAlchemy: stores all sessions in a single database via SQLAlchemy.
- Redis: stores all sessions in a single Redis data store.

Creating your own storage

The easiest way to create your own storage implementation is to use `MemorySession` as the base and check out how `SQLiteSession` or one of the community-maintained implementations work. You can find the relevant Python files under the `sessions` directory in Telethon.

After you have made your own implementation, you can add it to the community-maintained session implementation list above with a pull request.

1.8.2 SQLite Sessions and Heroku

You probably have a newer version of SQLite installed (>= 3.8.2). Heroku uses SQLite 3.7.9 which does not support `WITHOUT ROWID`. So, if you generated your session file on a system with SQLite >= 3.8.2 your session file will not work on Heroku's platform and will throw a corrupted schema error.

There are multiple ways to solve this, the easiest of which is generating a session file on your Heroku dyno itself. The most complicated is creating a custom buildpack to install SQLite >= 3.8.2.

Generating a SQLite Session File on a Heroku Dyno

Note: Due to Heroku's ephemeral filesystem all dynamically generated files not part of your applications buildpack or codebase are destroyed upon each restart.

Warning: Do not restart your application Dyno at any point prior to retrieving your session file. Constantly creating new session files from Telegram's API will result in a 24 hour rate limit ban.

Due to Heroku's ephemeral filesystem all dynamically generated files not part of your applications buildpack or codebase are destroyed upon each restart.

Using this scaffolded code we can start the authentication process:

```
client = TelegramClient('login.session', api_id, api_hash).start()
```

At this point your Dyno will crash because you cannot access stdin. Open your Dyno's control panel on the Heroku website and "Run console" from the "More" dropdown at the top right. Enter `bash` and wait for it to load.

You will automatically be placed into your applications working directory. So run your application `python app.py` and now you can complete the input requests such as "what is your phone number" etc.

Once you're successfully authenticated exit your application script with CTRL + C and `ls` to confirm `login.session` exists in your current directory. Now you can create a git repo on your account and commit `login.session` to that repo.

You cannot `ssh` into your Dyno instance because it has crashed, so unless you programatically upload this file to a server host this is the only way to get it off of your Dyno.

You now have a session file compatible with SQLite <= 3.8.2. Now you can programatically fetch this file from an external host (Firebase, S3 etc.) and login to your session using the following scaffolded code:

```
fileName, headers = urllib.request.urlretrieve(file_url, 'login.session')
client = TelegramClient(os.path.abspath(fileName), api_id, api_hash).start()
```

Note:

- `urlretrieve` will be depreciated, consider using `requests`.
- `file_url` represents the location of your file.

1.9 Update Modes

The library can run in four distinguishable modes:

- With no extra threads at all.
- With an extra thread that receives everything as soon as possible (default).
- With several worker threads that run your update handlers.
- A mix of the above.

Since this section is about updates, we'll describe the simplest way to work with them.

1.9.1 Using multiple workers

When you create your client, simply pass a number to the `update_workers` parameter:

```
client = TelegramClient('session', api_id, api_hash,
update_workers=2)
```

You can set any amount of workers you want. The more you put, the more update handlers that can be called "at the same time". One or two should suffice most of the time, since setting more will not make things run faster most of the times (actually, it could slow things down).

The next thing you want to do is to add a method that will be called when an Update arrives:

```
def callback(update):
    print('I received', update)

client.add_event_handler(callback)
# do more work here, or simply sleep!
```

That's it! This is the old way to listen for raw updates, with no further processing. If this feels annoying for you, remember that you can always use *Working with Updates* but maybe use this for some other cases.

Now let's do something more interesting. Every time an user talks to use, let's reply to them with the same text reversed:

```
from telethon.tl.types import UpdateShortMessage, PeerUser

def replier(update):
    if isinstance(update, UpdateShortMessage) and not update.out:
        client.send_message(PeerUser(update.user_id), update.message[::-1])


client.add_event_handler(replier)
input('Press enter to stop this!')
client.disconnect()
```

We only ask you one thing: don't keep this running for too long, or your contacts will go mad.

1.9.2 Spawning no worker at all

All the workers do is loop forever and poll updates from a queue that is filled from the `ReadThread`, responsible for reading every item off the network. If you only need a worker and the `MainThread` would be doing no other job, this is the preferred way. You can easily do the same as the workers like so:

```
while True:
    try:
        update = client.updates.poll()
        if not update:
            continue

        print('I received', update)
    except KeyboardInterrupt:
        break

client.disconnect()
```

Note that `poll` accepts a `timeout=` parameter, and it will return `None` if other thread got the update before you could or if the timeout expired, so it's important to check `if not update`.

This can coexist with the rest of `N` workers, or you can set it to `0` additional workers:

```
client = TelegramClient('session', api_id, api_hash,
update_workers=0)
```

You **must** set it to `0` (or higher), as it defaults to `None` and that has a different meaning. `None` workers means updates won't be processed *at all*, so you must set it to some integer value if you want `client.updates.poll()` to work.

1.9.3 Using the main thread instead the `ReadThread`

If you have no work to do on the `MainThread` and you were planning to have a `while True: sleep(1)`, don't do that. Instead, don't spawn the secondary `ReadThread` at all like so:

```
client = TelegramClient(
    ...
    spawn_read_thread=False
)
```

And then `.idle()` from the `MainThread`:

```
client.idle()
```

You can stop it with `Control+C`, and you can configure the signals to be used in a similar fashion to Python Telegram Bot.

As a complete example:

```
def callback(update):
    print('I received', update)

client = TelegramClient('session', api_id, api_hash,
                        update_workers=1, spawn_read_thread=False)

client.connect()
client.add_event_handler(callback)
client.idle()  # ends with Ctrl+C
```

This is the preferred way to use if you're simply going to listen for updates.

1.10 Working with messages

Note: These examples assume you have read *Accessing the Full API*.

1.10.1 Forwarding messages

This request is available as a friendly method through *client.forward_messages*, and can be used like shown below:

```
# If you only have the message IDs
client.forward_messages(
    entity,  # to which entity you are forwarding the messages
    message_ids,  # the IDs of the messages (or message) to forward
    from_entity  # who sent the messages?
)

# If you have ``Message`` objects
client.forward_messages(
    entity,  # to which entity you are forwarding the messages
    messages  # the messages (or message) to forward
)

# You can also do it manually if you prefer
```

```
from telethon.tl.functions.messages import ForwardMessagesRequest

messages = foo()  # retrieve a few messages (or even one, in a list)
from_entity = bar()
to_entity = baz()

client(ForwardMessagesRequest(
    from_peer=from_entity,  # who sent these messages?
    id=[msg.id for msg in messages],  # which are the messages?
    to_peer=to_entity  # who are we forwarding them to?
))
```

The named arguments are there for clarity, although they're not needed because they appear in order. You can obviously just wrap a single message on the list too, if that's all you have.

1.10.2 Searching Messages

Messages are searched through the obvious SearchRequest, but you may run into issues. A valid example would be:

```
from telethon.tl.functions.messages import SearchRequest
from telethon.tl.types import InputMessagesFilterEmpty

filter = InputMessagesFilterEmpty()
result = client(SearchRequest(
    peer=peer,       # On which chat/conversation
    q='query',       # What to search for
    filter=filter,   # Filter to use (maybe filter for media)
    min_date=None,   # Minimum date
    max_date=None,   # Maximum date
    offset_id=0,     # ID of the message to use as offset
    add_offset=0,    # Additional offset
    limit=10,        # How many results
    max_id=0,        # Maximum message ID
    min_id=0,        # Minimum message ID
    from_id=None     # Who must have sent the message (peer)
))
```

It's important to note that the optional parameter `from_id` could have been omitted (defaulting to `None`). Changing it to InputUserEmpty, as one could think to specify "no user", won't work because this parameter is a flag, and it being unspecified has a different meaning.

If one were to set `from_id=InputUserEmpty()`, it would filter messages from "empty" senders, which would likely match no users.

If you get a `ChatAdminRequiredError` on a channel, it's probably because you tried setting the `from_id` filter, and as the error says, you can't do that. Leave it set to `None` and it should work.

As with every method, make sure you use the right ID/hash combination for your `InputUser` or `InputChat`, or you'll likely run into errors like `UserIdInvalidError`.

1.10.3 Sending stickers

Stickers are nothing else than `files`, and when you successfully retrieve the stickers for a certain sticker set, all you will have are `handles` to these files. Remember, the files Telegram holds on their servers can be referenced through this pair of ID/hash (unique per user), and you need to use this handle when sending a "document" message. This working example will send yourself the very first sticker you have:

```
# Get all the sticker sets this user has
sticker_sets = client(GetAllStickersRequest(0))

# Choose a sticker set
sticker_set = sticker_sets.sets[0]

# Get the stickers for this sticker set
stickers = client(GetStickerSetRequest(
    stickerset=InputStickerSetID(
        id=sticker_set.id, access_hash=sticker_set.access_hash
    )
))

# Stickers are nothing more than files, so send that
client(SendMediaRequest(
    peer=client.get_me(),
    media=InputMediaDocument(
        id=InputDocument(
            id=stickers.documents[0].id,
            access_hash=stickers.documents[0].access_hash
        )
    )
))
```

1.11 Working with Chats and Channels

Note: These examples assume you have read *Accessing the Full API*.

1.11.1 Joining a chat or channel

Note that Chat are normal groups, and Channel are a special form of `Chat`, which can also be super-groups if their `megagroup` member is `True`.

1.11.2 Joining a public channel

Once you have the *entity* of the channel you want to join to, you can make use of the JoinChannelRequest to join such channel:

```
from telethon.tl.functions.channels import JoinChannelRequest
client(JoinChannelRequest(channel))

# In the same way, you can also leave such channel
from telethon.tl.functions.channels import LeaveChannelRequest
client(LeaveChannelRequest(input_channel))
```

For more on channels, check the channels namespace.

1.11.3 Joining a private chat or channel

If all you have is a link like this one: `https://t.me/joinchat/AAAAAFFszQPyPEZ7wgxLtd`, you already have enough information to join! The part after the `https://t.me/joinchat/`, this is, `AAAAAFFszQPyPEZ7wgxLtd` on this example, is the `hash` of the chat or channel. Now you can use ImportChatInviteRequest as follows:

```
from telethon.tl.functions.messages import ImportChatInviteRequest
updates = client(ImportChatInviteRequest('AAAAAEHbEkejzxUjAUCfYg'))
```

1.11.4 Adding someone else to such chat or channel

If you don't want to add yourself, maybe because you're already in, you can always add someone else with the AddChatUserRequest, which use is very straightforward, or InviteToChannelRequest for channels:

```
# For normal chats
from telethon.tl.functions.messages import AddChatUserRequest

# Note that ``user_to_add`` is NOT the name of the parameter.
# It's the user you want to add (``user_id=user_to_add``).
client(AddChatUserRequest(
    chat_id,
    user_to_add,
    fwd_limit=10  # Allow the user to see the 10 last messages
))

# For channels (which includes megagroups)
from telethon.tl.functions.channels import InviteToChannelRequest

client(InviteToChannelRequest(
    channel,
    [users_to_add]
))
```

1.11.5 Checking a link without joining

If you don't need to join but rather check whether it's a group or a channel, you can use the CheckChatInviteRequest, which takes in the hash of said channel or group.

1.11.6 Retrieving all chat members (channels too)

You can use *client.get_participants* to retrieve the participants (click it to see the relevant parameters). Most of the time you will just need `client.get_participants(entity)`.

This is what said method is doing behind the scenes as an example.

In order to get all the members from a mega-group or channel, you need to use GetParticipantsRequest. As we can see it needs an InputChannel, (passing the mega-group or channel you're going to use will work), and a mandatory ChannelParticipantsFilter. The closest thing to "no filter" is to simply use ChannelParticipantsSearch with an empty `'q'` string.

If we want to get *all* the members, we need to use a moving offset and a fixed limit:

```
from telethon.tl.functions.channels import GetParticipantsRequest
from telethon.tl.types import ChannelParticipantsSearch
from time import sleep

offset = 0
limit = 100
all_participants = []

while True:
    participants = client(GetParticipantsRequest(
        channel, ChannelParticipantsSearch(''), offset, limit,
        hash=0
    ))
    if not participants.users:
        break
    all_participants.extend(participants.users)
    offset += len(participants.users)
```

Note: If you need more than 10,000 members from a group you should use the mentioned `client.get_participants(..., aggressive=True)`. It will do some tricks behind the scenes to get as many entities as possible. Refer to issue 573 for more on this.

Note that GetParticipantsRequest returns ChannelParticipants, which may have more information you need (like the role of the participants, total count of members, etc.)

1.11.7 Recent Actions

"Recent actions" is simply the name official applications have given to the "admin log". Simply use GetAdminLogRequest for that, and you'll get AdminLogResults.events in return which in turn has the final .action.

1.11.8 Admin Permissions

Giving or revoking admin permissions can be done with the EditAdminRequest:

```
from telethon.tl.functions.channels import EditAdminRequest
from telethon.tl.types import ChannelAdminRights

# You need both the channel and who to grant permissions
# They can either be channel/user or input channel/input user.
#
# ChannelAdminRights is a list of granted permissions.
# Set to True those you want to give.
rights = ChannelAdminRights(
    post_messages=None,
    add_admins=None,
    invite_users=None,
    change_info=True,
    ban_users=None,
    delete_messages=True,
    pin_messages=True,
    invite_link=None,
    edit_messages=None
)
```

```
# Equivalent to:
#     rights = ChannelAdminRights(
#         change_info=True,
#         delete_messages=True,
#         pin_messages=True
#     )

# Once you have a ChannelAdminRights, invoke it
client(EditAdminRequest(channel, user, rights))

# User will now be able to change group info, delete other people's
# messages and pin messages.
```

Note: Thanks to @Kyle2142 for pointing out that you **cannot** set all parameters to `True` to give a user full permissions, as not all permissions are related to both broadcast channels/megagroups.

E.g. trying to set `post_messages=True` in a megagroup will raise an error. It is recommended to always use keyword arguments, and to set only the permissions the user needs. If you don't need to change a permission, it can be omitted (full list here).

1.11.9 Restricting Users

Similar to how you give or revoke admin permissions, you can edit the banned rights of an user through EditAdminRequest and its parameter ChannelBannedRights:

```
from telethon.tl.functions.channels import EditBannedRequest
from telethon.tl.types import ChannelBannedRights

from datetime import datetime, timedelta

# Restricting an user for 7 days, only allowing view/send messages.
#
# Note that it's "reversed". You must set to ``True`` the permissions
# you want to REMOVE, and leave as ``None`` those you want to KEEP.
rights = ChannelBannedRights(
    until_date=datetime.now() + timedelta(days=7),
    view_messages=None,
    send_messages=None,
    send_media=True,
    send_stickers=True,
    send_gifs=True,
    send_games=True,
    send_inline=True,
    embed_links=True
)

# The above is equivalent to
rights = ChannelBannedRights(
    until_date=datetime.now() + timedelta(days=7),
    send_media=True,
    send_stickers=True,
    send_gifs=True,
    send_games=True,
    send_inline=True,
```

```
    embed_links=True
)

client(EditBannedRequest(channel, user, rights))
```

1.11.10 Kicking a member

Telegram doesn't actually have a request to kick an user from a group. Instead, you need to restrict them so they can't see messages. Any date is enough:

```
from telethon.tl.functions.channels import EditBannedRequest
from telethon.tl.types import ChannelBannedRights

client(EditBannedRequest(channel, user, ChannelBannedRights(
    until_date=None,
    view_messages=True
)))
```

1.11.11 Increasing View Count in a Channel

It has been asked quite a few times (really, many), and while I don't understand why so many people ask this, the solution is to use GetMessagesViewsRequest, setting `increment=True`:

```
# Obtain `channel' through dialogs or through client.get_entity() or anyhow.
# Obtain `msg_ids' through `.get_message_history()` or anyhow. Must be a␣
↪list.

client(GetMessagesViewsRequest(
    peer=channel,
    id=msg_ids,
    increment=True
))
```

Note that you can only do this **once or twice a day** per account, running this in a loop will obviously not increase the views forever unless you wait a day between each iteration. If you run it any sooner than that, the views simply won't be increased.

1.12 Bots

Note: These examples assume you have read *Accessing the Full API*.

1.12.1 Talking to Inline Bots

You can query an inline bot, such as @VoteBot (note, *query*, not *interact* with a voting message), by making use of the GetInlineBotResultsRequest request:

```
from telethon.tl.functions.messages import GetInlineBotResultsRequest

bot_results = client(GetInlineBotResultsRequest(
    bot, user_or_chat, 'query', ''
))
```

And you can select any of their results by using SendInlineBotResultRequest:

```
from telethon.tl.functions.messages import SendInlineBotResultRequest

client(SendInlineBotResultRequest(
    get_input_peer(user_or_chat),
    obtained_query_id,
    obtained_str_id
))
```

1.12.2 Talking to Bots with special reply markup

To interact with a message that has a special reply markup, such as @VoteBot polls, you would use GetBotCallbackAnswerRequest:

```
from telethon.tl.functions.messages import GetBotCallbackAnswerRequest

client(GetBotCallbackAnswerRequest(
    user_or_chat,
    msg.id,
    data=msg.reply_markup.rows[wanted_row].buttons[wanted_button].data
))
```

It's a bit verbose, but it has all the information you would need to show it visually (button rows, and buttons within each row, each with its own data).

1.13 Projects using Telethon

This page lists some real world examples showcasing what can be built with the library.

Note: Do you have a project that uses the library or know of any that's not listed here? Feel free to leave a comment at issue 744 so it can be included in the next revision of the documentation!

1.13.1 telegram-export

Link / Author's website

A tool to download Telegram data (users, chats, messages, and media) into a database (and display the saved data).

1.13.2 mautrix-telegram

Link / Author's website

A Matrix-Telegram hybrid puppeting/relaybot bridge.

1.13.3 TelegramTUI

Link / Author's website

A Telegram client on your terminal.

1.14 Enabling Logging

Telethon makes use of the logging module, and you can enable it as follows:

```
import logging
logging.basicConfig(level=logging.DEBUG)
```

The library has the NullHandler added by default so that no log calls will be printed unless you explicitly enable it.

You can also use the module on your own project very easily:

```
import logging
logger = logging.getLogger(__name__)

logger.debug('Debug messages')
logger.info('Useful information')
logger.warning('This is a warning!')
```

If you want to enable `logging` for your project *but* use a different log level for the library:

```
import logging
logging.basicConfig(level=logging.DEBUG)
# For instance, show only warnings and above
logging.getLogger('telethon').setLevel(level=logging.WARNING)
```

1.15 Deleted, Limited or Deactivated Accounts

If you're from Iran or Russian, we have bad news for you. Telegram is much more likely to ban these numbers, as they are often used to spam other accounts, likely through the use of libraries like this one. The best advice we can give you is to not abuse the API, like calling many requests really quickly, and to sign up with these phones through an official application.

Telegram may also ban virtual (VoIP) phone numbers, as again, they're likely to be used for spam.

If you want to check if your account has been limited, simply send a private message to @SpamBot through Telegram itself. You should notice this by getting errors like `PeerFloodError`, which means you're limited, for instance, when sending a message to some accounts but not others.

For more discussion, please see issue 297.

1.16 RPC Errors

RPC stands for Remote Procedure Call, and when the library raises a `RPCError`, it's because you have invoked some of the API methods incorrectly (wrong parameters, wrong permissions, or even something went wrong on Telegram's server). All the errors are available in *telethon.errors package*, but some examples are:

- `FloodWaitError` (420), the same request was repeated many times. Must wait `.seconds` (you can access this parameter).
- `SessionPasswordNeededError`, if you have setup two-steps verification on Telegram.
- `CdnFileTamperedError`, if the media you were trying to download from a CDN has been altered.
- `ChatAdminRequiredError`, you don't have permissions to perform said operation on a chat or channel. Try avoiding filters, i.e. when searching messages.

The generic classes for different error codes are:

- `InvalidDCError` (303), the request must be repeated on another DC.
- `BadRequestError` (400), the request contained errors.
- `UnauthorizedError` (401), the user is not authorized yet.
- `ForbiddenError` (403), privacy violation error.
- `NotFoundError` (404), make sure you're invoking `Request`'s!

If the error is not recognised, it will only be an `RPCError`.

1.17 Philosophy

The intention of the library is to have an existing MTProto library existing with hardly any dependencies (indeed, wherever Python is available, you can run this library).

Being written in Python means that performance will be nowhere close to other implementations written in, for instance, Java, C++, Rust, or pretty much any other compiled language. However, the library turns out to actually be pretty decent for common operations such as sending messages, receiving updates, or other scripting. Uploading files may be notably slower, but if you would like to contribute, pull requests are appreciated!

If `libssl` is available on your system, the library will make use of it to speed up some critical parts such as encrypting and decrypting the messages. Files will notably be sent and downloaded faster.

The main focus is to keep everything clean and simple, for everyone to understand how working with MTProto and Telegram works. Don't be afraid to read the source, the code won't bite you! It may prove useful when using the library on your own use cases.

1.18 API Status

In an attempt to help everyone who works with the Telegram API, the library will by default report all *Remote Procedure Call* errors to RPC PWRTelegram, a public database anyone can query, made by Daniil. All the information sent is a `GET` request with the error code, error message and method used.

If you still would like to opt out, you can disable this feature by setting `client.session.report_errors = False`. However Daniil would really thank you if you helped him (and everyone) by keeping it on!

1.18.1 Querying the API status

The API is accessed through `GET` requests, which can be made for instance through `curl`. A JSON response will be returned.

All known errors and their description:

```
curl https://rpc.pwrtelegram.xyz/?all
```

Error codes for a specific request:

```
curl https://rpc.pwrtelegram.xyz/?for=messages.sendMessage
```

Number of `RPC_CALL_FAIL`:

```
curl https://rpc.pwrtelegram.xyz/?rip  # last hour
curl https://rpc.pwrtelegram.xyz/?rip=$(time()-60)  # last minute
```

Description of errors:

```
curl https://rpc.pwrtelegram.xyz/?description_for=SESSION_REVOKED
```

Code of a specific error:

```
curl https://rpc.pwrtelegram.xyz/?code_for=STICKERSET_INVALID
```

1.19 Test Servers

To run Telethon on a test server, use the following code:

```
client = TelegramClient(None, api_id, api_hash)
client.session.set_dc(dc_id, '149.154.167.40', 80)
```

You can check your `'test ip'` on https://my.telegram.org.

You should set `None` session so to ensure you're generating a new authorization key for it (it would fail if you used a session where you had previously connected to another data center).

Note that port 443 might not work, so you can try with 80 instead.

Once you're connected, you'll likely be asked to either sign in or sign up. Remember anyone can access the phone you choose, so don't store sensitive data here.

Valid phone numbers are `99966XYYYY`, where `X` is the `dc_id` and `YYYY` is any number you want, for example, `1234` in `dc_id = 2` would be `9996621234`. The code sent by Telegram will be `dc_id` repeated five times, in this case, `22222` so we can hardcode that:

```
client = TelegramClient(None, api_id, api_hash)
client.session.set_dc(2, '149.154.167.40', 80)
client.start(phone='9996621234', code_callback=lambda: '22222')
```

1.20 Project Structure

1.20.1 Main interface

The library itself is under the `telethon/` directory. The `__init__.py` file there exposes the main `TelegramClient`, a class that servers as a nice interface with the most commonly used methods on Telegram such as sending messages, retrieving the message history, handling updates, etc.

The `TelegramClient` inherits the `TelegramBareClient`. The later is basically a pruned version of the `TelegramClient`, which knows basic stuff like `.invoke()`'ing requests, downloading files, or switching between data centers. This is primary to keep the method count per class and file low and manageable.

Both clients make use of the `network/mtproto_sender.py`. The `MtProtoSender` class handles packing requests with the `salt`, `id`, `sequence`, etc., and also handles how to process responses (i.e. pong, RPC errors). This class communicates through Telegram via its `.connection` member.

The `Connection` class uses a `extensions/tcp_client`, a C#-like `TcpClient` to ease working with sockets in Python. All the `TcpClient` know is how to connect through TCP and writing/reading from the socket with optional cancel.

The `Connection` class bundles up all the connections modes and sends and receives the messages accordingly (TCP full, obfuscated, intermediate...).

1.20.2 Auto-generated code

The files under `telethon_generator/` are used to generate the code that gets placed under `telethon/tl/`. The `TLGenerator` takes in a `.tl` file, and spits out the generated classes which represent, as Python classes, the request and types defined in the `.tl` file. It also constructs an index so that they can be imported easily.

1.21 Coding Style

Basically, make it **readable**, while keeping the style similar to the code of whatever file you're working on.

Also note that not everyone has 4K screens for their primary monitors, so please try to stick to the 80-columns limit. This makes it easy to `git diff` changes from a terminal before committing changes. If the line has to be long, please don't exceed 120 characters.

For the commit messages, please make them *explanatory*. Not only they're helpful to troubleshoot when certain issues could have been introduced, but they're also used to construct the change log once a new version is ready.

If you don't know enough Python, I strongly recommend reading Dive Into Python 3, available online for free. For instance, remember to do `if x is None` or `if x is not None` instead `if x == None`!

1.22 Understanding the Type Language

Telegram's Type Language (also known as TL, found on `.tl` files) is a concise way to define what other programming languages commonly call classes or structs.

Every definition is written as follows for a Telegram object is defined as follows:

```
name#id argument_name:argument_type = CommonType
```

This means that in a single line you know what the `TLObject` name is. You know it's unique ID, and you know what arguments it has. It really isn't that hard to write a generator for generating code to any platform!

The generated code should also be able to *encode* the `TLObject` (let this be a request or a type) into bytes, so they can be sent over the network. This isn't a big deal either, because you know how the `TLObject`'s are made, and how the types should be serialized.

You can either write your own code generator, or use the one this library provides, but please be kind and keep some special mention to this project for helping you out.

This is only a introduction. The `TL` language is not *that* easy. But it's not that hard either. You're free to sniff the `telethon_generator/` files and learn how to parse other more complex lines, such as `flags` (to indicate things that may or may not be written at all) and `vector`'s.

1.23 Tips for Porting the Project

If you're going to use the code on this repository to guide you, please be kind and don't forget to mention it helped you!

You should start by reading the source code on the first release of the project, and start creating a `MtProtoSender`. Once this is made, you should write by hand the code to authenticate on the Telegram's server, which are some steps required to get the key required to talk to them. Save it somewhere! Then, simply mimic, or reinvent other parts of the code, and it will be ready to go within a few days.

Good luck!

1.24 Telegram API in Other Languages

Telethon was made for **Python**, and as far as I know, there is no *exact* port to other languages. However, there *are* other implementations made by awesome people (one needs to be awesome to understand the official Telegram documentation) on several languages (even more Python too), listed below:

1.24.1 C

Possibly the most well-known unofficial open source implementation out there by @vysheng, tgl, and its console client telegram-cli. Latest development has been moved to BitBucket.

1.24.2 C++

The newest (and official) library, written from scratch, is called tdlib and is what the Telegram X uses. You can find more information in the official documentation, published here.

1.24.3 JavaScript

@zerobias is working on telegram-mtproto, a work-in-progress JavaScript library installable via npm.

1.24.4 Kotlin

Kotlogram is a Telegram implementation written in Kotlin (one of the official languages for Android) by @badoualy, currently as a beta– yet working.

1.24.5 PHP

A PHP implementation is also available thanks to @danog and his MadelineProto project, with a very nice online documentation too.

1.24.6 Python

A fairly new (as of the end of 2017) Telegram library written from the ground up in Python by @delivrance and his Pyrogram library. There isn't really a reason to pick it over Telethon and it'd be kinda sad to see you go, but it would be nice to know what you miss from each other library in either one so both can improve.

1.24.7 Rust

Yet another work-in-progress implementation, this time for Rust thanks to @JuanPotato under the fancy name of Vail.

1.25 Changelog (Version History)

This page lists all the available versions of the library, in chronological order. You should read this when upgrading the library to know where your code can break, and where it can take advantage of new goodies!

List of All Versions

1.25.1 Several bug fixes (v0.18.2)

Published at 2018/03/27

Just a few bug fixes before they become too many.

Additions

- Getting an entity by its positive ID should be enough, regardless of their type (whether it's an `User`, a `Chat` or a `Channel`). Although wrapping them inside a `Peer` is still recommended, it's not necessary.
- New `client.edit_2fa` function to change your Two Factor Authentication settings.
- `.stringify()` and string representation for custom `Dialog/Draft`.

Bug fixes

- Some bug regarding `.get_input_peer`.
- `events.ChatAction` wasn't picking up all the pins.
- `force_document=True` was being ignored for albums.
- Now you're able to send `Photo` and `Document` as files.
- Wrong access to a member on chat forbidden error for `.get_participants`. An empty list is returned instead.
- `me/self` check for `.get[_input]_entity` has been moved up so if someone has "me" or "self" as their name they won't be retrieved.

1.25.2 Iterator methods (v0.18.1)

Published at 2018/03/17

All the `.get_` methods in the `TelegramClient` now have a `.iter_` counterpart, so you can do operations while retrieving items from them. For instance, you can `client.iter_dialogs()` and `break` once you find what you're looking for instead fetching them all at once.

Another big thing, you can get entities by just their positive ID. This may cause some collisions (although it's very unlikely), and you can (should) still be explicit about the type you want. However, it's a lot more convenient and less confusing.

Breaking changes

- The library only offers the default `SQLiteSession` again. See *Session Files* for more on how to use a different storage from now on.

Additions

- Events now override `__str__` and implement `.stringify()`, just like every other `TLObject` does.
- `events.ChatAction` now has `respond()`, `reply()` and `delete()` for the message that triggered it.
- `client.iter_participants()` (and its `client.get_participants()` counterpart) now expose the `filter` argument, and the returned users also expose the `.participant` they are.
- You can now use `client.remove_event_handler()` and `client.list_event_handlers()` similar how you could with normal updates.
- New properties on `events.NewMessage`, like `.video_note` and `.gif` to access only specific types of documents.
- The `Draft` class now exposes `.text` and `.raw_text`, as well as a new `Draft.send()` to send it.

Bug fixes

- `MessageEdited` was ignoring `NewMessage` constructor arguments.
- Fixes for `Event.delete_messages` which wouldn't handle `MessageService`.
- Bot API style IDs not working on `client.get_input_entity()`.
- `client.download_media()` didn't support `PhotoSize`.

Enhancements

- Less RPC are made when accessing the `.sender` and `.chat` of some events (mostly those that occur in a channel).
- You can send albums larger than 10 items (they will be sliced for you), as well as mixing normal files with photos.
- `TLObject` now have Python type hints.

Internal changes

- Several documentation corrections.
- `client.get_dialogs()` is only called once again when an entity is not found to avoid flood waits.

1.25.3 Sessions overhaul (v0.18)

Published at 2018/03/04

Scheme layer used: 75

The `Session`'s have been revisited thanks to the work of **@tulir** and they now use an ABC so you can easily implement your own!

The default will still be a `SQLiteSession`, but you might want to use the new `AlchemySessionContainer` if you need. Refer to the section of the documentation on *Session Files* for more.

Breaking changes

- `events.MessageChanged` doesn't exist anymore. Use the new `events.MessageEdited` and `events.MessageDeleted` instead.

Additions

- The mentioned addition of new session types.
- You can omit the event type on `client.add_event_handler` to use `Raw`.
- You can `raise StopPropagation` of events if you added several of them.
- `.get_participants()` can now get up to 90,000 members from groups with 100,000 if when `aggressive=True`, "bypassing" Telegram's limit.
- You now can access `NewMessage.Event.pattern_match`.
- Multiple captions are now supported when sending albums.
- `client.send_message()` has an optional `file=` parameter, so you can do `events.reply(file='/path/to/photo.jpg')` and similar.
- Added `.input_` versions to `events.ChatAction`.
- You can now access the public `.client` property on `events`.
- New `client.forward_messages`, with its own wrapper on `events`, called `event.forward_to(...)`.

Bug fixes

- Silly bug regarding `client.get_me(input_peer=True)`.
- `client.send_voice_note()` was missing some parameters.
- `client.send_file()` plays better with streams now.
- Incoming messages from bots weren't working with whitelists.
- Markdown's URL regex was not accepting newlines.
- Better attempt at joining background update threads.
- Use the right peer type when a marked integer ID is provided.

Internal changes

- Resolving `events.Raw` is now a no-op.
- Logging calls in the `TcpClient` to spot errors.
- `events` resolution is postponed until you are successfully connected, so you can attach them before starting the client.
- When an entity is not found, it is searched in *all* dialogs. This might not always be desirable but it's more comfortable for legitimate uses.
- Some non-persisting properties from the `Session` have been moved out.

1.25.4 Further easing library usage (v0.17.4)

Published at 2018/02/24

Some new things and patches that already deserved their own release.

Additions

- New `pattern` argument to `NewMessage` to easily filter messages.
- New `.get_participants()` convenience method to get members from chats.
- `.send_message()` now accepts a `Message` as the `message` parameter.
- You can now `.get_entity()` through exact name match instead username.
- Raise `ProxyConnectionError` instead looping forever so you can `except` it on your own code and behave accordingly.

Bug fixes

- `.parse_username` would fail with `www.` or a trailing slash.
- `events.MessageChanged` would fail with `UpdateDeleteMessages`.
- You can now send `b'byte strings'` directly as files again.
- `.send_file()` was not respecting the original captions when passing another message (or media) as the file.
- Downloading media from a different data center would always log a warning for the first time.

Internal changes

- Use `req_pq_multi` instead `req_pq` when generating `auth_key`.
- You can use `.get_me(input_peer=True)` if all you need is your self ID.
- New addition to the interactive client example to show peer information.
- Avoid special casing `InputPeerSelf` on some `NewMessage` events, so you can always safely rely on `.sender` to get the right ID.

1.25.5 New small convenience functions (v0.17.3)

Published at 2018/02/18

More bug fixes and a few others addition to make events easier to use.

Additions

- Use `hachoir` to extract video and audio metadata before upload.
- New `.add_event_handler`, `.add_update_handler` now deprecated.

Bug fixes

- `bot_token` wouldn't work on `.start()`, and changes to `password` (now it will ask you for it if you don't provide it, as docstring hinted).
- `.edit_message()` was ignoring the formatting (e.g. markdown).
- Added missing case to the `NewMessage` event for normal groups.
- Accessing the `.text` of the `NewMessage` event was failing due to a bug with the markdown unparser.

Internal changes

- `libssl` is no longer an optional dependency. Use `cryptg` instead, which you can find on https://github.com/Lonami/cryptg.

1.25.6 New small convenience functions (v0.17.2)

Published at 2018/02/15

Primarily bug fixing and a few welcomed additions.

Additions

- New convenience `.edit_message()` method on the `TelegramClient`.
- New `.edit()` and `.delete()` shorthands on the `NewMessage` event.
- Default to markdown parsing when sending and editing messages.
- Support for inline mentions when sending and editing messages. They work like inline urls (e.g. `[text](@username)`) and also support the Bot-API style (see here).

Bug fixes

- Periodically send `GetStateRequest` automatically to keep the server sending updates even if you're not invoking any request yourself.
- HTML parsing was failing due to not handling surrogates properly.
- `.sign_up` was not accepting `int` codes.
- Whitelisting more than one chat on `events` wasn't working.

- Video files are sent as a video by default unless `force_document`.

Internal changes

- More `logging` calls to help spot some bugs in the future.
- Some more logic to retrieve input entities on events.
- Clarified a few parts of the documentation.

1.25.7 Updates as Events (v0.17.1)

Published at 2018/02/09

Of course there was more work to be done regarding updates, and it's here! The library comes with a new `events` module (which you will often import as `from telethon import TelegramClient, events`). This are pretty much all the additions that come with this version change, but they are a nice addition. Refer to *Working with Updates* to get started with events.

1.25.8 Trust the Server with Updates (v0.17)

Published at 2018/02/03

The library trusts the server with updates again. The library will *not* check for duplicates anymore, and when the server kicks us, it will run `GetStateRequest` so the server starts sending updates again (something it wouldn't do unless you invoked something, it seems). But this update also brings a few more changes!

Additions

- `TLObject`'s override `__eq__` and `__ne__`, so you can compare them.
- Added some missing cases on `.get_input_entity()` and peer functions.
- `obj.to_dict()` now has a `'_'` key with the type used.
- `.start()` can also sign up now.
- More parameters for `.get_message_history()`.
- Updated list of RPC errors.
- HTML parsing thanks to **@tulir**! It can be used similar to markdown: `client.send_message(..., parse_mode='html')`.

Enhancements

- `client.send_file()` now accepts `Message`'s and `MessageMedia`'s as the `file` parameter.
- Some documentation updates and fixed to clarify certain things.
- New exact match feature on https://lonamiwebs.github.io/Telethon.
- Return as early as possible from `.get_input_entity()` and similar, to avoid penalizing you for doing this right.

Bug fixes

- `.download_media()` wouldn't accept a `Document` as parameter.
- The SQLite is now closed properly on disconnection.
- IPv6 addresses shouldn't use square braces.
- Fix regarding `.log_out()`.
- The time offset wasn't being used (so having wrong system time would cause the library not to work at all).

1.25.9 New `.resolve()` method (v0.16.2)

Published at 2018/01/19

The `TLObject`'s (instances returned by the API and `Request`'s) have now acquired a new `.resolve()` method. While this should be used by the library alone (when invoking a request), it means that you can now use `Peer` types or even usernames where a `InputPeer` is required. The object now has access to the `client`, so that it can fetch the right type if needed, or access the session database. Furthermore, you can reuse requests that need "autocast" (e.g. you put User but `InputPeer` was needed), since `.resolve()` is called when invoking. Before, it was only done on object construction.

Additions

- Album support. Just pass a list, tuple or any iterable to `.send_file()`.

Enhancements

- `.start()` asks for your phone only if required.
- Better file cache. All files under 10MB, once uploaded, should never be needed to be re-uploaded again, as the sent media is cached to the session.

Bug fixes

- `setup.py` now calls `gen_tl` when installing the library if needed.

Internal changes

- The mentioned `.resolve()` to perform "autocast", more powerful.
- Upload and download methods are no longer part of `TelegramBareClient`.
- Reuse `.on_response()`, `.__str__` and `.stringify()`. Only override `.on_response()` if necessary (small amount of cases).
- Reduced "autocast" overhead as much as possible. You shouldn't be penalized if you've provided the right type.

1.25.10 MtProto 2.0 (v0.16.1)

Published at 2018/01/11

Scheme layer used: 74

The library is now using MtProto 2.0! This shouldn't really affect you as an end user, but at least it means the library will be ready by the time MtProto 1.0 is deprecated.

Additions

- New `.start()` method, to make the library avoid boilerplate code.
- `.send_file` accepts a new optional `thumbnail` parameter, and returns the `Message` with the sent file.

Bug fixes

- The library uses again only a single connection. Less updates are be dropped now, and the performance is even better than using temporary connections.
- `without rowid` will only be used on the `*.session` if supported.
- Phone code hash is associated with phone, so you can change your mind when calling `.sign_in()`.

Internal changes

- File cache now relies on the hash of the file uploaded instead its path, and is now persistent in the `*.session` file. Report any bugs on this!
- Clearer error when invoking without being connected.
- Markdown parser doesn't work on bytes anymore (which makes it cleaner).

1.25.11 Sessions as sqlite databases (v0.16)

Published at 2017/12/28

In the beginning, session files used to be pickle. This proved to be bad as soon as one wanted to add more fields. For this reason, they were migrated to use JSON instead. But this proved to be bad as soon as one wanted to save things like entities (usernames, their ID and hash), so now it properly uses sqlite3, which has been well tested, to save the session files! Calling `.get_input_entity` using a `username` no longer will need to fetch it first, so it's really 0 calls again. Calling `.get_entity` will always fetch the most up to date version.

Furthermore, nearly everything has been documented, thus preparing the library for Read the Docs (although there are a few things missing I'd like to polish first), and the logging are now better placed.

Breaking changes

- `.get_dialogs()` now returns a **single list** instead a tuple consisting of a **custom class** that should make everything easier to work with.
- `.get_message_history()` also returns a **single list** instead a tuple, with the `Message` instances modified to make them more convenient.

Both lists have a `.total` attribute so you can still know how many dialogs/messages are in total.

Additions

- The mentioned use of `sqlite3` for the session file.
- `.get_entity()` now supports lists too, and it will make as little API calls as possible if you feed it `InputPeer` types. Usernames will always be resolved, since they may have changed.
- `.set_proxy()` method, to avoid having to create a new `TelegramClient`.
- More `date` types supported to represent a date parameter.

Bug fixes

- Empty strings weren't working when they were a flag parameter (e.g., setting no last name).
- Fix invalid assertion regarding flag parameters as well.
- Avoid joining the background thread on disconnect, as it would be `None` due to a race condition.
- Correctly handle `None` dates when downloading media.
- `.download_profile_photo` was failing for some channels.
- `.download_media` wasn't handling `Photo`.

Internal changes

- `date` was being serialized as local date, but that was wrong.
- `date` was being represented as a `float` instead of an `int`.
- `.tl` parser wasn't stripping inline comments.
- Removed some redundant checks on `update_state.py`.
- Use a synchronized queue instead a hand crafted version.
- Use signed integers consistently (e.g. `salt`).
- Always read the corresponding `TLObject` from API responses, except for some special cases still.
- A few more `except` low level to correctly wrap errors.
- More accurate exception types.
- `invokeWithLayer(initConnection(X))` now wraps every first request after `.connect()`.

As always, report if you have issues with some of the changes!

1.25.12 IPv6 support (v0.15.5)

Published at 2017/11/16

Scheme layer used: 73

It's here, it has come! The library now **supports IPv6**! Just pass `use_ipv6=True` when creating a `TelegramClient`. Note that I could *not* test this feature because my machine doesn't have IPv6 setup. If you know IPv6 works in your machine but the library doesn't, please refer to #425.

Additions

- IPv6 support.
- New method to extract the text surrounded by `MessageEntity`'s, in the `extensions.markdown` module.

Enhancements

- Markdown parsing is Done Right.
- Reconnection on failed invoke. Should avoid "number of retries reached 0" (#270).
- Some missing autocast to `Input*` types.
- The library uses the `NullHandler` for `logging` as it should have always done.
- `TcpClient.is_connected()` is now more reliable.

Bug fixes

- Getting an entity using their phone wasn't actually working.
- Full entities aren't saved unless they have an `access_hash`, to avoid some `None` errors.
- `.get_message_history` was failing when retrieving items that had messages forwarded from a channel.

1.25.13 General enhancements (v0.15.4)

Published at 2017/11/04

Scheme layer used: 72

This update brings a few general enhancements that are enough to deserve a new release, with a new feature: beta **markdown-like parsing** for `.send_message()`!

Additions

- `.send_message()` supports `parse_mode='md'` for **Markdown**! It works in a similar fashion to the official clients (defaults to double underscore/asterisk, like `**this**`). Please report any issues with emojies or enhancements for the parser!
- New `.idle()` method so your main thread can do useful job (listen for updates).
- Add missing `.to_dict()`, `__str__` and `.stringify()` for `TLMessage` and `MessageContainer`.

Bug fixes

- The list of known peers could end "corrupted" and have users with `access_hash=None`, resulting in `struct` error for it not being an integer. You shouldn't encounter this issue anymore.
- The warning for "added update handler but no workers set" wasn't actually working.
- `.get_input_peer` was ignoring a case for `InputPeerSelf`.
- There used to be an exception when logging exceptions (whoops) on update handlers.

- "Downloading contacts" would produce strange output if they had semicolons (`;`) in their name.
- Fix some cyclic imports and installing dependencies from the `git` repository.
- Code generation was using f-strings, which are only supported on Python 3.6.

Internal changes

- The `auth_key` generation has been moved from `.connect()` to `.invoke()`. There were some issues were `.connect()` failed and the `auth_key` was `None` so this will ensure to have a valid `auth_key` when needed, even if `BrokenAuthKeyError` is raised.
- Support for higher limits on `.get_history()` and `.get_dialogs()`.
- Much faster integer factorization when generating the required `auth_key`. Thanks @delivrance for making me notice this, and for the pull request.

1.25.14 Bug fixes with updates (v0.15.3)

Published at 2017/10/20

Hopefully a very ungrateful bug has been removed. When you used to invoke some request through update handlers, it could potentially enter an infinite loop. This has been mitigated and it's now safe to invoke things again! A lot of updates were being dropped (all those gzipped), and this has been fixed too.

More bug fixes include a correct parsing of certain TLObjects thanks to @stek29, and some wrong calls that would cause the library to crash thanks to @andr-04, and the `ReadThread` not re-starting if you were already authorized.

Internally, the `.to_bytes()` function has been replaced with `__bytes__` so now you can do `bytes(tlobject)`.

1.25.15 Bug fixes and new small features (v0.15.2)

Published at 2017/10/14

This release primarly focuses on a few bug fixes and enhancements. Although more stuff may have broken along the way.

Enhancements

- You will be warned if you call `.add_update_handler` with no `update_workers`.
- New customizable threshold value on the session to determine when to automatically sleep on flood waits. See `client.session.flood_sleep_threshold`.
- New `.get_drafts()` method with a custom `Draft` class by @JosXa.
- Join all threads when calling `.disconnect()`, to assert no dangling thread is left alive.
- Larger chunk when downloading files should result in faster downloads.
- You can use a callable key for the `EntityDatabase`, so it can be any filter you need.

Bug fixes

- `.get_input_entity` was failing for IDs and other cases, also making more requests than it should.
- Use `basename` instead `abspath` when sending a file. You can now also override the attributes.
- `EntityDatabase.__delitem__` wasn't working.
- `.send_message()` was failing with channels.
- `.get_dialogs(limit=None)` should now return all the dialogs correctly.
- Temporary fix for abusive duplicated updates.

Internal changes

- MsgsAck is now sent in a container rather than its own request.
- `.get_input_photo` is now used in the generated code.
- `.process_entities` was being called from more places than only `__call__`.
- `MtProtoSender` now relies more on the generated code to read responses.

1.25.16 Custom Entity Database (v0.15.1)

Published at 2017/10/05

The main feature of this release is that Telethon now has a custom database for all the entities you encounter, instead depending on `@lru_cache` on the `.get_entity()` method.

The `EntityDatabase` will, by default, **cache** all the users, chats and channels you find in memory for as long as the program is running. The session will, by default, save all key-value pairs of the entity identifiers and their hashes (since Telegram may send an ID that it thinks you already know about, we need to save this information).

You can **prevent** the `EntityDatabase` from saving users by setting `client.session.entities.enabled = False`, and prevent the `Session` from saving input entities at all by setting `client.session.save_entities = False`. You can also clear the cache for a certain user through `client.session.entities.clear_cache(entity=None)`, which will clear all if no entity is given.

Additions

- New method to `.delete_messages()`.
- New `ChannelPrivateError` class.

Enhancements

- `.sign_in` accepts phones as integers.
- Changing the IP to which you connect to is as simple as `client.session.server_address = 'ip'`, since now the server address is always queried from the session.

Bug fixes

- `.get_dialogs()` doesn't fail on Windows anymore, and returns the right amount of dialogs.
- `GeneralProxyError` should be passed to the main thread again, so that you can handle it.

1.25.17 Updates Overhaul Update (v0.15)

Published at 2017/10/01

After hundreds of lines changed on a major refactor, *it's finally here*. It's the **Updates Overhaul Update**; let's get right into it!

Breaking changes

- `.create_new_connection()` is gone for good. No need to deal with this manually since new connections are now handled on demand by the library itself.

Enhancements

- You can **invoke** requests from **update handlers**. And **any other thread**. A new temporary will be made, so that you can be sending even several requests at the same time!
- **Several worker threads** for your updates! By default, `None` will spawn. I recommend you to work with `update_workers=4` to get started, these will be polling constantly for updates.
- You can also change the number of workers at any given time.
- The library can now run **in a single thread** again, if you don't need to spawn any at all. Simply set `spawn_read_thread=False` when creating the `TelegramClient`!
- You can specify `limit=None` on `.get_dialogs()` to get **all** of them[1].
- **Updates are expanded**, so you don't need to check if the update has `.updates` or an inner `.update` anymore.
- All `InputPeer` entities are **saved in the session** file, but you can disable this by setting `save_entities=False`.
- New `.get_input_entity` method, which makes use of the above feature. You **should use this** when a request needs a `InputPeer`, rather than the whole entity (although both work).
- Assert that either all or None dependent-flag parameters are set before sending the request.
- Phone numbers can have dashes, spaces, or parenthesis. They'll be removed before making the request.
- You can override the phone and its hash on `.sign_in()`, if you're creating a new `TelegramClient` on two different places.

Bug fixes

- `.log_out()` was consuming all retries. It should work just fine now.
- The session would fail to load if the `auth_key` had been removed manually.
- `Updates.check_error` was popping wrong side, although it's been completely removed.
- `ServerError`'s will be **ignored**, and the request will immediately be retried.

- Cross-thread safety when saving the session file.
- Some things changed on a matter of when to reconnect, so please report any bugs!

Internal changes

- `TelegramClient` is now only an abstraction over the `TelegramBareClient`, which can only do basic things, such as invoking requests, working with files, etc. If you don't need any of the abstractions the `TelegramClient`, you can now use the `TelegramBareClient` in a much more comfortable way.
- `MtProtoSender` is not thread-safe, but it doesn't need to be since a new connection will be spawned when needed.
- New connections used to be cached and then reused. Now only their sessions are saved, as temporary connections are spawned only when needed.
- Added more RPC errors to the list.

[1]: Broken due to a condition which should had been the opposite (sigh), fixed 4 commits ahead on https://github.com/LonamiWebs/Telethon/commit/62ea77cbeac7c42bfac85aa8766a1b5b35e3a76c.

That's pretty much it, although there's more work to be done to make the overall experience of working with updates *even better*. Stay tuned!

1.25.18 Serialization bug fixes (v0.14.2)

Published at 2017/09/29

Bug fixes

- **Important**, related to the serialization. Every object or request that had to serialize a `True/False` type was always being serialized as `false`!
- Another bug that didn't allow you to leave as `None` flag parameters that needed a list has been fixed.

Internal changes

- Other internal changes include a somewhat more readable `.to_bytes()` function and pre-computing the flag instead using bit shifting. The `TLObject.constructor_id` has been renamed to `TLObject.CONSTRUCTOR_ID`, and `.subclass_of_id` is also uppercase now.

1.25.19 Farewell, BinaryWriter (v0.14.1)

Published at 2017/09/28

Version `v0.14` had started working on the new `.to_bytes()` method to dump the `BinaryWriter` and its usage on the `.on_send()` when serializing TLObjects, and this release finally removes it. The speed up when serializing things to bytes should now be over twice as fast wherever it's needed.

Bug fixes

- This version is again compatible with Python 3.x versions **below 3.5** (there was a method call that was Python 3.5 and above).

Internal changes

- Using proper classes (including the generated code) for generating authorization keys and to write out `TLMessage`'s.

1.25.20 Several requests at once and upload compression (v0.14)

Published at 2017/09/27

New major release, since I've decided that these two features are big enough:

Additions

- Requests larger than 512 bytes will be **compressed through gzip**, and if the result is smaller, this will be uploaded instead.
- You can now send **multiple requests at once**, they're simply `*var_args` on the `.invoke()`. Note that the server doesn't guarantee the order in which they'll be executed!

Internally, another important change. The `.on_send` function on the `TLObjects` is **gone**, and now there's a new `.to_bytes()`. From my tests, this has always been over twice as fast serializing objects, although more replacements need to be done, so please report any issues.

Enhancements

- Implemented `.get_input_media` helper methods. Now you can even use another message as input media!

Bug fixes

- Downloading media from CDNs wasn't working (wrong access to a parameter).
- Correct type hinting.
- Added a tiny sleep when trying to perform automatic reconnection.
- Error reporting is done in the background, and has a shorter timeout.
- `setup.py` used to fail with wrongly generated code.

1.25.21 Quick fix-up (v0.13.6)

Published at 2017/09/23

Before getting any further, here's a quick fix-up with things that should have been on `v0.13.5` but were missed. Specifically, the **timeout when receiving** a request will now work properly.

Some other additions are a tiny fix when **handling updates**, which was ignoring some of them, nicer `__str__` and `.stringify()` methods for the `TLObject`'s, and not stopping the `ReadThread` if you try invoking something there (now it simply returns `None`).

1.25.22 Attempts at more stability (v0.13.5)

Published at 2017/09/23

Yet another update to fix some bugs and increase the stability of the library, or, at least, that was the attempt!

This release should really **improve the experience with the background thread** that the library starts to read things from the network as soon as it can, but I can't spot every use case, so please report any bug (and as always, minimal reproducible use cases will help a lot).

Bug fixes

- `setup.py` was failing on Python < 3.5 due to some imports.
- Duplicated updates should now be ignored.
- `.send_message` would crash in some cases, due to having a typo using the wrong object.
- `"socket is None"` when calling `.connect()` should not happen anymore.
- `BrokenPipeError` was still being raised due to an incorrect order on the `try/except` block.

Enhancements

- **Type hinting** for all the generated `Request`'s and `TLObjects`! IDEs like PyCharm will benefit from this.
- `ProxyConnectionError` should properly be passed to the main thread for you to handle.
- The background thread will only be started after you're authorized on Telegram (i.e. logged in), and several other attempts at polishing the experience with this thread.
- The `Connection` instance is only created once now, and reused later.
- Calling `.connect()` should have a better behavior now (like actually *trying* to connect even if we seemingly were connected already).
- `.reconnect()` behavior has been changed to also be more consistent by making the assumption that we'll only reconnect if the server has disconnected us, and is now private.

Internal changes

- `TLObject.__repr__` doesn't show the original TL definition anymore, it was a lot of clutter. If you have any complaints open an issue and we can discuss it.
- Internally, the `'+'` from the phone number is now stripped, since it shouldn't be included.
- Spotted a new place where `BrokenAuthKeyError` would be raised, and it now is raised there.

1.25.23 More bug fixes and enhancements (v0.13.4)

Published at 2017/09/18

Additions

- `TelegramClient` now exposes a `.is_connected()` method.
- Initial authorization on a new data center will retry up to 5 times by default.
- Errors that couldn't be handled on the background thread will be raised on the next call to `.invoke()` or `updates.poll()`.

Bug fixes

- Now you should be able to sign in even if you have `process_updates=True` and no previous session.
- Some errors and methods are documented a bit clearer.
- `.send_message()` could randomly fail, as the returned type was not expected.
- `TimeoutError` is now ignored, since the request will be retried up to 5 times by default.
- "-404" errors (`BrokenAuthKeyError`'s) are now detected when first connecting to a new data center.
- `BufferError` is handled more gracefully, in the same way as `InvalidCheckSumError`'s.
- Attempt at fixing some "NoneType has no attribute..." errors (with the `.sender`).

Internal changes

- Calling `GetConfigRequest` is now made less often.
- The `initial_query` parameter from `.connect()` is gone, as it's not needed anymore.
- Renamed `all_tlobjects.layer` to `all_tlobjects.LAYER` (since it's a constant).
- The message from `BufferError` is now more useful.

1.25.24 Bug fixes and enhancements (v0.13.3)

Published at 2017/09/14

Bug fixes

- **Reconnection** used to fail because it tried invoking things from the `ReadThread`.
- Inferring **random ids** for `ForwardMessagesRequest` wasn't working.
- Downloading media from **CDNs** failed due to having forgotten to remove a single line.
- `TcpClient.close()` now has a **"threading.Lock"**, so `NoneType has no close()` should not happen.
- New **workaround** for `msg seqno too low/high`. Also, both `Session.id/seq` are not saved anymore.

Enhancements

- **Request will be retried** up to 5 times by default rather than failing on the first attempt.
- `InvalidChecksumError`'s are now **ignored** by the library.
- `TelegramClient.get_entity()` is now **public**, and uses the `@lru_cache()` decorator.
- New method to **''.send_voice_note()''**'s.
- Methods to send message and media now support a **''reply_to'' parameter**.
- `.send_message()` now returns the **full message** which was just sent.

1.25.25 New way to work with updates (v0.13.2)

Published at 2017/09/08

This update brings a new way to work with updates, and it's begging for your **feedback**, or better names or ways to do what you can do now.

Please refer to the wiki/Usage Modes for an in-depth description on how to work with updates now. Notice that you cannot invoke requests from within handlers anymore, only the `v.0.13.1` patch allowed you to do so.

Bug fixes

- Periodic pings are back.
- The username regex mentioned on `UsernameInvalidError` was invalid, but it has now been fixed.
- Sending a message to a phone number was failing because the type used for a request had changed on layer 71.
- CDN downloads weren't working properly, and now a few patches have been applied to ensure more reliability, although I couldn't personally test this, so again, report any feedback.

1.25.26 Invoke other requests from within update callbacks (v0.13.1)

Published at 2017/09/04

> **Warning:** This update brings some big changes to the update system, so please read it if you work with them!

A silly "bug" which hadn't been spotted has now been fixed. Now you can invoke other requests from within your update callbacks. However **this is not advised**. You should post these updates to some other thread, and let that thread do the job instead. Invoking a request from within a callback will mean that, while this request is being invoked, no other things will be read.

Internally, the generated code now resides under a *lot* less files, simply for the sake of avoiding so many unnecessary files. The generated code is not meant to be read by anyone, simply to do its job.

Unused attributes have been removed from the `TLObject` class too, and `.sign_up()` returns the user that just logged in in a similar way to `.sign_in()` now.

1.25.27 Connection modes (v0.13)

Published at 2017/09/04

Scheme layer used: 71

The purpose of this release is to denote a big change, now you can connect to Telegram through different **connection modes**. Also, a **second thread** will *always* be started when you connect a `TelegramClient`, despite whether you'll be handling updates or ignoring them, whose sole purpose is to constantly read from the network.

The reason for this change is as simple as *"reading and writing shouldn't be related"*. Even when you're simply ignoring updates, this way, once you send a request you will only need to read the result for the request. Whatever Telegram sent before has already been read and outside the buffer.

Additions

- The mentioned different connection modes, and a new thread.
- You can modify the `Session` attributes through the `TelegramClient` constructor (using `**kwargs`).
- `RPCError`'s now belong to some request you've made, which makes more sense.
- `get_input_*` now handles `None` (default) parameters more gracefully (it used to crash).

Enhancements

- The low-level socket doesn't use a handcrafted timeout anymore, which should benefit by avoiding the arbitrary `sleep(0.1)` that there used to be.
- `TelegramClient.sign_in` will call `.send_code_request` if no `code` was provided.

Deprecation

- `.sign_up` does *not* take a `phone` argument anymore. Change this or you will be using `phone` as `code`, and it will fail! The definition looks like `def sign_up(self, code, first_name, last_name='')`.
- The old `JsonSession` finally replaces the original `Session` (which used pickle). If you were overriding any of these, you should only worry about overriding `Session` now.

1.25.28 Added verification for CDN file (v0.12.2)

Published at 2017/08/28

Since the Content Distributed Network (CDN) is not handled by Telegram itself, the owners may tamper these files. Telegram sends their sha256 sum for clients to implement this additional verification step, which now the library has. If any CDN has altered the file you're trying to download, `CdnFileTamperedError` will be raised to let you know.

Besides this. `TLObject.stringify()` was showing bytes as lists (now fixed) and RPC errors are reported by default:

> In an attempt to help everyone who works with the Telegram API, Telethon will by default report all Remote Procedure Call errors to PWRTelegram, a public database anyone can query, made by Daniil. All the information sent is a GET request with the error code, error message and method used.

Note: If you still would like to opt out, simply set `client.session.report_errors = False` to disable this feature. However Daniil would really thank you if you helped him (and everyone) by keeping it on!

1.25.29 CDN support (v0.12.1)

Published at 2017/08/24

The biggest news for this update are that downloading media from CDN's (you'll often encounter this when working with popular channels) now **works**.

Bug fixes

- The method used to download documents crashed because two lines were swapped.
- Determining the right path when downloading any file was very weird, now it's been enhanced.
- The `.sign_in()` method didn't support integer values for the code! Now it does again.

Some important internal changes are that the old way to deal with RSA public keys now uses a different module instead the old strange hand-crafted version.

Hope the new, super simple `README.rst` encourages people to use Telethon and make it better with either suggestions, or pull request. Pull requests are *super* appreciated, but showing some support by leaving a star also feels nice .

1.25.30 Newbie friendly update (v0.12)

Published at 2017/08/22

Scheme layer used: 70

This update is overall an attempt to make Telethon a bit more user friendly, along with some other stability enhancements, although it brings quite a few changes.

Breaking changes

- The `TelegramClient` methods `.send_photo_file()`, `.send_document_file()` and `.send_media_file()` are now a **single method** called `.send_file()`. It's also important to note that the **order** of the parameters has been **swapped**: first to *who* you want to send it, then the file itself.
- The same applies to `.download_msg_media()`, which has been renamed to `.download_media()`. The method now supports a `Message` itself too, rather than only `Message.media`. The specialized `.download_photo()`, `.download_document()` and `.download_contact()` still exist, but are private.

Additions

- Updated to **layer 70**!
- Both downloading and uploading now support **stream-like objects**.
- A lot **faster initial connection** if `sympy` is installed (can be installed through `pip`).

- `libssl` will also be used if available on your system (likely on Linux based systems). This speed boost should also apply to uploading and downloading files.
- You can use a **phone number** or an **username** for methods like `.send_message()`, `.send_file()`, and all the other quick-access methods provided by the `TelegramClient`.

Bug fixes

- Crashing when migrating to a new layer and receiving old updates should not happen now.
- `InputPeerChannel` is now casted to `InputChannel` automtically too.
- `.get_new_msg_id()` should now be thread-safe. No promises.
- Logging out on macOS caused a crash, which should be gone now.
- More checks to ensure that the connection is flagged correctly as either connected or not.

Note: Downloading files from CDN's will **not work** yet (something new that comes with layer 70).

That's it, any new idea or suggestion about how to make the project even more friendly is highly appreciated.

Note: Did you know that you can pretty print any result Telegram returns (called `TLObject`'s) by using their `.stringify()` function? Great for debugging!

1.25.31 get_input_* now works with vectors (v0.11.5)

Published at 2017/07/11

Quick fix-up of a bug which hadn't been encountered until now. Auto-cast by using `get_input_*` now works.

1.25.32 get_input_* everywhere (v0.11.4)

Published at 2017/07/10

For some reason, Telegram doesn't have enough with the InputPeer. There also exist InputChannel and InputUser! You don't have to worry about those anymore, it's handled internally now.

Besides this, every Telegram object now features a new default `.__str__` look, and also a .stringify() method to pretty format them, if you ever need to inspect them.

The library now uses the DEBUG level everywhere, so no more warnings or information messages if you had logging enabled.

The `no_webpage` parameter from `.send_message` has been renamed to `link_preview` for clarity, so now it does the opposite (but has a clearer intention).

1.25.33 Quick .send_message() fix (v0.11.3)

Published at 2017/07/05

A very quick follow-up release to fix a tiny bug with `.send_message()`, no new features.

1.25.34 Callable TelegramClient (v0.11.2)

Published at 2017/07/04

Scheme layer used: 68

There is a new preferred way to **invoke requests**, which you're encouraged to use:

```
# New!
result = client(SomeRequest())

# Old.
result = client.invoke(SomeRequest())
```

Existing code will continue working, since the old `.invoke()` has not been deprecated.

When you `.create_new_connection()`, it will also handle `FileMigrateError`'s for you, so you don't need to worry about those anymore.

Bugs fixes

- Fixed some errors when installing Telethon via `pip` (for those using either source distributions or a Python version 3.5).
- `ConnectionResetError` didn't flag sockets as closed, but now it does.

On a more technical side, `msg_id`'s are now more accurate.

1.25.35 Improvements to the updates (v0.11.1)

Published at 2017/06/24

Receiving new updates shouldn't miss any anymore, also, periodic pings are back again so it should work on the long run.

On a different order of things, `.connect()` also features a timeout. Notice that the `timeout=` is **not** passed as a **parameter** anymore, and is instead specified when creating the `TelegramClient`.

Bug fixes

- Fixed some name class when a request had a `.msg_id` parameter.
- The correct amount of random bytes is now used in DH request
- Fixed `CONNECTION_APP_VERSION_EMPTY` when using temporary sessions.
- Avoid connecting if already connected.

1.25.36 Support for parallel connections (v0.11)

Published at 2017/06/16

This update brings a lot of changes, so it would be nice if you could **read the whole change log**!

Breaking changes

- Every Telegram error has now its **own class**, so it's easier to fine-tune your `except`'s.
- Markdown parsing is **not part** of Telethon itself anymore, although there are plans to support it again through a some external module.
- The `.list_sessions()` has been moved to the `Session` class instead.
- The `InteractiveTelegramClient` is **not** shipped with `pip` anymore.

Additions

- A new, more **lightweight class** has been added. The `TelegramBareClient` is now the base of the normal `TelegramClient`, and has the most basic features.
- New method to `.create_new_connection()`, which can be ran **in parallel** with the original connection. This will return the previously mentioned `TelegramBareClient` already connected.
- Any file object can now be used to download a file (for instance, a `BytesIO()` instead a file name).
- Vales like `random_id` are now **automatically inferred**, so you can save yourself from the hassle of writing `generate_random_long()` everywhere. Same applies to `.get_input_peer()`, unless you really need the extra performance provided by skipping one `if` if called manually.
- Every type now features a new `.to_dict()` method.

Bug fixes

- Received errors are acknowledged to the server, so they don't happen over and over.
- Downloading media on different data centers is now up to **x2 faster**, since there used to be an `InvalidDCError` for each file part tried to be downloaded.
- Lost messages are now properly skipped.
- New way to handle the **result of requests**. The old `ValueError` "*The previously sent request must be resent. However, no request was previously sent (possibly called from a different thread).*" *should* not happen anymore.

Internal changes

- Some fixes to the `JsonSession`.
- Fixed possibly crashes if trying to `.invoke()` a `Request` while `.reconnect()` was being called on the `UpdatesThread`.
- Some improvements on the `TcpClient`, such as not switching between blocking and non-blocking sockets.
- The code now uses ASCII characters only.
- Some enhancements to `.find_user_or_chat()` and `.get_input_peer()`.

1.25.37 JSON session file (v0.10.1)

Published at 2017/06/07

This version is primarily for people to **migrate** their `.session` files, which are *pickled*, to the new *JSON* format. Although slightly slower, and a bit more vulnerable since it's plain text, it's a lot more resistant to upgrades.

Warning: You **must** upgrade to this version before any higher one if you've used Telethon v0.10. If you happen to upgrade to an higher version, that's okay, but you will have to manually delete the `*.session` file, and logout from that session from an official client.

Additions

- New `.get_me()` function to get the **current** user.
- `.is_user_authorized()` is now more reliable.
- New nice button to copy the `from telethon.tl.xxx.yyy import Yyy` on the online documentation.
- **More error codes** added to the `errors` file.

Enhancements

- Everything on the documentation is now, theoretically, **sorted alphabetically**.
- No second thread is spawned unless one or more update handlers are added.

1.25.38 Full support for different DCs and ++stable (v0.10)

Published at 2017/06/03

Working with **different data centers** finally *works*! On a different order of things, **reconnection** is now performed automatically every time Telegram decides to kick us off their servers, so now Telethon can really run **forever and ever**! In theory.

Enhancements

- **Documentation** improvements, such as showing the return type.
- The `msg_id too low/high` error should happen **less often**, if any.
- Sleeping on the main thread is **not done anymore**. You will have to `except FloodWaitError`'s.
- You can now specify your *own application version*, device model, system version and language code.
- Code is now more *pythonic* (such as making some members private), and other internal improvements (which affect the **updates thread**), such as using `logger` instead a bare `print()` too.

This brings Telethon a whole step closer to `v1.0`, though more things should preferably be changed.

1.25.39 Stability improvements (v0.9.1)

Published at 2017/05/23

Telethon used to crash a lot when logging in for the very first time. The reason for this was that the reconnection (or dead connections) were not handled properly. Now they are, so you should be able to login directly, without needing to delete the `*.session` file anymore. Notice that downloading from a different DC is still a WIP.

Enhancements

- Updates thread is only started after a successful login.
- Files meant to be ran by the user now use **shebangs** and proper permissions.
- In-code documentation now shows the returning type.
- **Relative import** is now used everywhere, so you can rename `telethon` to anything else.
- **Dead connections** are now **detected** instead entering an infinite loop.
- **Sockets** can now be **closed** (and re-opened) properly.
- Telegram decided to update the layer 66 without increasing the number. This has been fixed and now we're up-to-date again.

1.25.40 General improvements (v0.9)

Published at 2017/05/19

Scheme layer used: 66

Additions

- The **documentation**, available online here, has a new search bar.
- Better **cross-thread safety** by using `threading.Event`.
- More improvements for running Telethon during a **long period of time**.

Bug fixes

- **Avoid a certain crash on login** (occurred if an unexpected object ID was received).
- Avoid crashing with certain invalid UTF-8 strings.
- Avoid crashing on certain terminals by using known ASCII characters where possible.
- The `UpdatesThread` is now a daemon, and should cause less issues.
- Temporary sessions didn't actually work (with `session=None`).

Internal changes

- `.get_dialogs(count=` was renamed to `.get_dialogs(limit=`.

1.25.41 Bot login and proxy support (v0.8)

Published at 2017/04/14

Additions

- **Bot login**, thanks to @JuanPotato for hinting me about how to do it.
- **Proxy support**, thanks to @exzhawk for implementing it.
- **Logging support**, used by passing `--telethon-log=DEBUG` (or `INFO`) as a command line argument.

Bug fixes

- Connection fixes, such as avoiding connection until `.connect()` is explicitly invoked.
- Uploading big files now works correctly.
- Fix uploading big files.
- Some fixes on the updates thread, such as correctly sleeping when required.

1.25.42 Long-run bug fix (v0.7.1)

Published at 2017/02/19

If you're one of those who runs Telethon for a long time (more than 30 minutes), this update by @strayge will be great for you. It sends periodic pings to the Telegram servers so you don't get disconnected and you can still send and receive updates!

1.25.43 Two factor authentication (v0.7)

Published at 2017/01/31

Scheme layer used: 62

If you're one of those who love security the most, these are good news. You can now use two factor authentication with Telethon too! As internal changes, the coding style has been improved, and you can easily use custom session objects, and various little bugs have been fixed.

1.25.44 Updated pip version (v0.6)

Published at 2016/11/13

Scheme layer used: 57

This release has no new major features. However, it contains some small changes that make using Telethon a little bit easier. Now those who have installed Telethon via `pip` can also take advantage of changes, such as less bugs, creating empty instances of `TLObjects`, specifying a timeout and more!

1.25.45 Ready, pip, go! (v0.5)

Published at 2016/09/18

Telethon is now available as a **'Python package <https://pypi.python.org/pypi?name=Telethon>'__**! Those are really exciting news (except, sadly, the project structure had to change *a lot* to be able to do that; but hopefully it won't need to change much more, any more!)

Not only that, but more improvements have also been made: you're now able to both **sign up** and **logout**, watch a pretty "Uploading/Downloading... x%" progress, and other minor changes which make using Telethon **easier**.

1.25.46 Made InteractiveTelegramClient cool (v0.4)

Published at 2016/09/12

Yes, really cool! I promise. Even though this is meant to be a *library*, that doesn't mean it can't have a good *interactive client* for you to try the library out. This is why now you can do many, many things with the `InteractiveTelegramClient`:

- **List dialogs** (chats) and pick any you wish.
- **Send any message** you like, text, photos or even documents.
- **List** the **latest messages** in the chat.
- **Download** any message's media (photos, documents or even contacts!).
- **Receive message updates** as you talk (i.e., someone sent you a message).

It actually is an usable-enough client for your day by day. You could even add `libnotify` and pop, you're done! A great cli-client with desktop notifications.

Also, being able to download and upload media implies that you can do the same with the library itself. Did I need to mention that? Oh, and now, with even less bugs! I hope.

1.25.47 Media revolution and improvements to update handling! (v0.3)

Published at 2016/09/11

Telegram is more than an application to send and receive messages. You can also **send and receive media**. Now, this implementation also gives you the power to upload and download media from any message that contains it! Nothing can now stop you from filling up all your disk space with all the photos! If you want to, of course.

1.25.48 Handle updates in their own thread! (v0.2)

Published at 2016/09/10

This version handles **updates in a different thread** (if you wish to do so). This means that both the low level `TcpClient` and the not-so-low-level `MtProtoSender` are now multi-thread safe, so you can use them with more than a single thread without worrying!

This also implies that you won't need to send a request to **receive an update** (is someone typing? did they send me a message? has someone gone offline?). They will all be received **instantly**.

Some other cool examples of things that you can do: when someone tells you "*Hello*", you can automatically reply with another "*Hello*" without even needing to type it by yourself :)

However, be careful with spamming!! Do **not** use the program for that!

1.25.49 First working alpha version! (v0.1)

Published at 2016/09/06

Scheme layer used: 55

There probably are some bugs left, which haven't yet been found. However, the majority of code works and the application is already usable! Not only that, but also uses the latest scheme as of now *and* handles way better the errors. This tag is being used to mark this release as stable enough.

1.26 Wall of Shame

This project has an issues section for you to file **issues** whenever you encounter any when working with the library. Said section is **not** for issues on *your* program but rather issues with Telethon itself.

If you have not made the effort to 1. read through the docs and 2. look for the method you need, you will end up on the Wall of Shame, i.e. all issues labeled "RTFM":

> **rtfm** Literally "Read The F–king Manual"; a term showing the frustration of being bothered with questions so trivial that the asker could have quickly figured out the answer on their own with minimal effort, usually by reading readily-available documents. People who say"RTFM!" might be considered rude, but the true rude ones are the annoying people who take absolutely no self-responibility and expect to have all the answers handed to them personally.
>
> *"Damn, that's the twelveth time that somebody posted this question to the messageboard today! RTFM, already!"*
>
> *by Bill M. July 27, 2004*

If you have indeed read the docs, and have tried looking for the method, and yet you didn't find what you need, **that's fine**. Telegram's API can have some obscure names at times, and for this reason, there is a "question" label with questions that are okay to ask. Just state what you've tried so that we know you've made an effort, or you'll go to the Wall of Shame.

Of course, if the issue you're going to open is not even a question but a real issue with the library (thankfully, most of the issues have been that!), you won't end up here. Don't worry.

1.26.1 Current winner

The current winner is issue 213:

Issue:

Answer:

Fig. 1.1: Winner issue

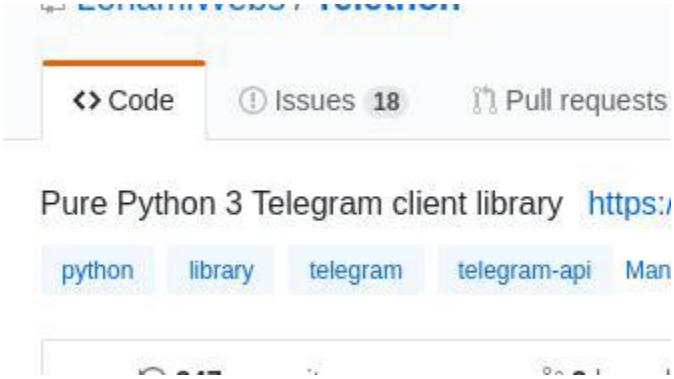

Fig. 1.2: Winner issue answer

1.27 telethon

1.27.1 telethon package

telethon.telegram_client module

class telethon.telegram_client.**TelegramClient**(*session*, *api_id*, *api_hash*, *connection_mode=<ConnectionMode.TCP_FULL: 1>*, *use_ipv6=False*, *proxy=None*, *update_workers=None*, *timeout=datetime.timedelta(0, 10)*, *spawn_read_thread=True*, *report_errors=True*, ***kwargs*)

Bases: *telethon.telegram_bare_client.TelegramBareClient*

Initializes the Telegram client with the specified API ID and Hash.

Args:

session (str | telethon.sessions.abstract.Session, None): The file name of the session file to be used if a string is given (it may be a full path), or the Session instance to be used otherwise. If it's None, the session will not be saved, and you should call *log_out()* when you're done.

api_id (int | str): The API ID you obtained from https://my.telegram.org.

api_hash (str): The API ID you obtained from https://my.telegram.org.

connection_mode (`ConnectionMode`, optional): The connection mode to be used when creating a new connection to the servers. Defaults to the `TCP_FULL` mode. This will only affect how messages are sent over the network and how much processing is required before sending them.

use_ipv6 (`bool`, optional): Whether to connect to the servers through IPv6 or not. By default this is `False` as IPv6 support is not too widespread yet.

proxy (`tuple` | `dict`, optional): A tuple consisting of `(socks.SOCKS5, 'host', port)`. See https://github.com/Anorov/PySocks#usage-1 for more.

update_workers (`int`, optional): If specified, represents how many extra threads should be spawned to handle incoming updates, and updates will be kept in memory until they are processed. Note that you must set this to at least `0` if you want to be able to process updates through `updates.poll()`.

timeout (`int` | `float` | `timedelta`, optional): The timeout to be used when receiving responses from the network. Defaults to 5 seconds.

spawn_read_thread (`bool`, optional): Whether to use an extra background thread or not. Defaults to `True` so receiving items from the network happens instantly, as soon as they arrive. Can still be disabled if you want to run the library without any additional thread.

report_errors (`bool`, optional): Whether to report RPC errors or not. Defaults to `True`, see *API Status* for more information.

Kwargs: Some extra parameters are required when establishing the first connection. These are are (along with their default values):

```
device_model     = platform.node()
system_version   = platform.system()
app_version      = TelegramClient.__version__
lang_code        = 'en'
system_lang_code = lang_code
```

`add_event_handler`(*callback*, *event=None*)

Registers the given callback to be called on the specified event.

Args:

callback (`callable`): The callable function accepting one parameter to be used.

event (`_EventBuilder` | `type`, optional): The event builder class or instance to be used, for instance `events.NewMessage`.

If left unspecified, `events.Raw` (the `Update` objects with no further processing) will be passed instead.

`add_update_handler`(*handler*)

`delete_messages`(*entity*, *message_ids*, *revoke=True*)

Deletes a message from a chat, optionally "for everyone".

Args:

entity (`entity`): From who the message will be deleted. This can actually be `None` for normal chats, but **must** be present for channels and megagroups.

message_ids (`list` | `int` | Message): The IDs (or ID) or messages to be deleted.

revoke (`bool`, optional): Whether the message should be deleted for everyone or not. By default it has the opposite behaviour of official clients, and it will delete the message for everyone. This has no effect on channels or megagroups.

Returns: The AffectedMessages.

download_file(*input_location*, *file*, *part_size_kb=None*, *file_size=None*, *progress_callback=None*)
Downloads the given input location to a file.

Args:

input_location (InputFileLocation): The file location from which the file will be downloaded.

file (str | file): The output file path, directory, or stream-like object. If the path exists and is a file, it will be overwritten.

part_size_kb (int, optional): Chunk size when downloading files. The larger, the less requests will be made (up to 512KB maximum).

file_size (int, optional): The file size that is about to be downloaded, if known. Only used if `progress_callback` is specified.

progress_callback (callable, optional): A callback function accepting two parameters: `(downloaded bytes, total)`. Note that the `total` is the provided `file_size`.

download_media(*message*, *file=None*, *progress_callback=None*)
Downloads the given media, or the media from a specified Message.

message (Message | Media): The media or message containing the media that will be downloaded.

file (str | file, optional): The output file path, directory, or stream-like object. If the path exists and is a file, it will be overwritten.

progress_callback (callable, optional): A callback function accepting two parameters: `(received bytes, total)`.

Returns: `None` if no media was provided, or if it was Empty. On success the file path is returned since it may differ from the one given.

download_profile_photo(*entity*, *file=None*, *download_big=True*)
Downloads the profile photo of the given entity (user/chat/channel).

Args:

entity (entity): From who the photo will be downloaded.

file (str | file, optional): The output file path, directory, or stream-like object. If the path exists and is a file, it will be overwritten.

download_big (bool, optional): Whether to use the big version of the available photos.

Returns: `None` if no photo was provided, or if it was Empty. On success the file path is returned since it may differ from the one given.

edit_2fa(*current_password=None*, *new_password=None*, *hint=''*, *email=None*)
Changes the 2FA settings of the logged in user, according to the passed parameters. Take note of the parameter explanations.

Has no effect if both current and new password are omitted.

current_password (str, optional): The current password, to authorize changing to `new_password`. Must be set if changing existing 2FA settings. Must **not** be set if 2FA is currently disabled. Passing this by itself will remove 2FA (if correct).

new_password (str, optional): The password to set as 2FA. If 2FA was already enabled, `current_password` **must** be set. Leaving this blank or `None` will remove the password.

hint (str, optional): Hint to be displayed by Telegram when it asks for 2FA. Leaving unspecified is highly discouraged. Has no effect if `new_password` is not set.

email (str, optional): Recovery and verification email. Raises `EmailUnconfirmedError` if value differs from current one, and has no effect if `new_password` is not set.

Returns: `True` if successful, `False` otherwise.

edit_message(*entity*, *message_id*, *message=None*, *parse_mode='md'*, *link_preview=True*)
Edits the given message ID (to change its contents or disable preview).

Args:

entity (entity): From which chat to edit the message.

message_id (str): The ID of the message (or `Message` itself) to be edited.

message (str, optional): The new text of the message.

parse_mode (str, optional): Can be 'md' or 'markdown' for markdown-like parsing (default), or 'htm' or 'html' for HTML-like parsing. If `None` or any other false-y value is provided, the message will be sent with no formatting.

link_preview (bool, optional): Should the link preview be shown?

Raises: `MessageAuthorRequiredError` if you're not the author of the message but try editing it anyway.

`MessageNotModifiedError` if the contents of the message were not modified at all.

Returns: The edited Message.

forward_messages(*entity*, *messages*, *from_peer=None*)
Forwards the given message(s) to the specified entity.

Args:

entity (entity): To which entity the message(s) will be forwarded.

messages (list | int | Message): The message(s) to forward, or their integer IDs.

from_peer (entity): If the given messages are integer IDs and not instances of the `Message` class, this *must* be specified in order for the forward to work.

Returns: The list of forwarded Message, or a single one if a list wasn't provided as input.

get_dialogs(**args*, ***kwargs*)
Same as *iter_dialogs()*, but returns a list instead with an additional `.total` attribute on the list.

get_drafts()
Same as *iter_drafts()*, but returns a list instead.

get_entity(*entity*)
Turns the given entity into a valid Telegram user or chat.

entity (str | int | Peer | InputPeer): The entity (or iterable of entities) to be transformed. If it's a string which can be converted to an integer or starts with '+' it will be resolved as if it were a phone number.

If it doesn't start with '+' or starts with a '@' it will be be resolved from the username. If no exact match is returned, an error will be raised.

If the entity is an integer or a Peer, its information will be returned through a call to self.get_input_peer(entity).

If the entity is neither, and it's not a TLObject, an error will be raised.

Returns: User, Chat or Channel corresponding to the input entity. A list will be returned if more than one was given.

get_input_entity(*peer*)
Turns the given peer into its input entity version. Most requests use this kind of InputUser, InputChat and so on, so this is the most suitable call to make for those cases.

entity (str | int | Peer | InputPeer): The integer ID of an user or otherwise either of a PeerUser, PeerChat or PeerChannel, for which to get its `Input*` version.

If this `Peer` hasn't been seen before by the library, the top dialogs will be loaded and their entities saved to the session file (unless this feature was disabled explicitly).

If in the end the access hash required for the peer was not found, a ValueError will be raised.

Returns: InputPeerUser, InputPeerChat or InputPeerChannel.

get_me(*input_peer=False*)
Gets "me" (the self user) which is currently authenticated, or None if the request fails (hence, not authenticated).

Args:

input_peer (bool, optional): Whether to return the InputPeerUser version or the normal User. This can be useful if you just need to know the ID of yourself.

Returns: Your own User.

get_message_history(**args*, ***kwargs*)

get_messages(**args*, ***kwargs*)
Same as *iter_messages()*, but returns a list instead with an additional `.total` attribute on the list.

get_participants(**args*, ***kwargs*)
Same as *iter_participants()*, but returns a list instead with an additional `.total` attribute on the list.

iter_dialogs(*limit=None*, *offset_date=None*, *offset_id=0*, *offset_peer=<telethon.tl.types.InputPeerEmpty object>*, *_total=None*)
Returns an iterator over the dialogs, yielding 'limit' at most. Dialogs are the open "chats" or conversations with other people.

Args:

limit (int | None): How many dialogs to be retrieved as maximum. Can be set to `None` to retrieve all dialogs. Note that this may take whole minutes if you have hundreds of dialogs, as Telegram will tell the library to slow down through a `FloodWaitError`.

offset_date (datetime, optional): The offset date to be used.

offset_id (int, optional): The message ID to be used as an offset.

offset_peer (InputPeer, optional): The peer to be used as an offset.

_total (list, optional): A single-item list to pass the total parameter by reference.

Yields: Instances of *telethon.tl.custom.dialog.Dialog*.

iter_drafts()
Iterator over all open draft messages.

Instances of *telethon.tl.custom.draft.Draft* are yielded. You can call *telethon.tl.custom.draft.Draft.set_message* to change the message or *telethon.tl.custom.draft.Draft.delete* among other things.

iter_messages(*entity*, *limit=20*, *offset_date=None*, *offset_id=0*, *max_id=0*, *min_id=0*, *add_offset=0*, *batch_size=100*, *wait_time=None*, *_total=None*)
Iterator over the message history for the specified entity.

Args:

entity (`entity`): The entity from whom to retrieve the message history.

limit (`int` | `None`, optional): Number of messages to be retrieved. Due to limitations with the API retrieving more than 3000 messages will take longer than half a minute (or even more based on previous calls). The limit may also be `None`, which would eventually return the whole history.

offset_date (`datetime`): Offset date (messages *previous* to this date will be retrieved). Exclusive.

offset_id (`int`): Offset message ID (only messages *previous* to the given ID will be retrieved). Exclusive.

max_id (`int`): All the messages with a higher (newer) ID or equal to this will be excluded

min_id (`int`): All the messages with a lower (older) ID or equal to this will be excluded.

add_offset (`int`): Additional message offset (all of the specified offsets + this offset = older messages).

batch_size (`int`): Messages will be returned in chunks of this size (100 is the maximum). While it makes no sense to modify this value, you are still free to do so.

wait_time (`int`): Wait time between different GetHistoryRequest. Use this parameter to avoid hitting the `FloodWaitError` as needed. If left to `None`, it will default to 1 second only if the limit is higher than 3000.

_total (`list`, optional): A single-item list to pass the total parameter by reference.

Yields: Instances of Message with extra attributes:

- `.sender` = entity of the sender.
- `.fwd_from.sender` = if fwd_from, who sent it originally.
- `.fwd_from.channel` = if fwd_from, original channel.
- `.to` = entity to which the message was sent.

Notes: Telegram's flood wait limit for GetHistoryRequest seems to be around 30 seconds per 3000 messages, therefore a sleep of 1 second is the default for this limit (or above). You may need an higher limit, so you're free to set the `batch_size` that you think may be good.

`iter_participants`(*entity*, *limit=None*, *search=''*, *filter=None*, *aggressive=False*, *_total=None*)
Iterator over the participants belonging to the specified chat.

Args:

entity (`entity`): The entity from which to retrieve the participants list.

limit (`int`): Limits amount of participants fetched.

search (`str`, optional): Look for participants with this string in name/username.

filter (ChannelParticipantsFilter, optional): The filter to be used, if you want e.g. only admins Note that you might not have permissions for some filter. This has no effect for normal chats or users.

aggressive (`bool`, optional): Aggressively looks for all participants in the chat in order to get more than 10,000 members (a hard limit imposed by Telegram). Note that this might take a long time (over 5 minutes), but is able to return over 90,000 participants on groups with 100,000 members.

This has no effect for groups or channels with less than 10,000 members, or if a `filter` is given.

_total (`list`, optional): A single-item list to pass the total parameter by reference.

Yields: The User objects returned by GetParticipantsRequest with an additional `.participant` attribute which is the matched ChannelParticipant type for channels/megagroups or ChatParticipants for normal chats.

list_event_handlers()
Lists all added event handlers, returning a list of pairs consisting of (callback, event).

list_update_handlers()

log_out()
Logs out Telegram and deletes the current `*.session` file.

Returns: `True` if the operation was successful.

on(*event*)
Decorator helper method around add_event_handler().

Args:

event (_EventBuilder | type): The event builder class or instance to be used, for instance `events.NewMessage`.

remove_event_handler(*callback*, *event=None*)
Inverse operation of *add_event_handler()*.

If no event is given, all events for this callback are removed. Returns how many callbacks were removed.

remove_update_handler(*handler*)

send_code_request(*phone*, *force_sms=False*)
Sends a code request to the specified phone number.

Args:

phone (str | int): The phone to which the code will be sent.

force_sms (bool, optional): Whether to force sending as SMS.

Returns: An instance of SentCode.

send_file(*entity*, *file*, *caption=''*, *force_document=False*, *progress_callback=None*, *reply_to=None*, *attributes=None*, *thumb=None*, *allow_cache=True*, *parse_mode='md'*, ***kwargs*)
Sends a file to the specified entity.

Args:

entity (entity): Who will receive the file.

file (str | bytes | file | media): The path of the file, byte array, or stream that will be sent. Note that if a byte array or a stream is given, a filename or its type won't be inferred, and it will be sent as an "unnamed application/octet-stream".

Subsequent calls with the very same file will result in immediate uploads, unless `.clear_file_cache()` is called.

Furthermore the file may be any media (a message, document, photo or similar) so that it can be resent without the need to download and re-upload it again.

If a list or similar is provided, the files in it will be sent as an album in the order in which they appear, sliced in chunks of 10 if more than 10 are given.

caption (str, optional): Optional caption for the sent media message.

force_document (bool, optional): If left to `False` and the file is a path that ends with the extension of an image file or a video file, it will be sent as such. Otherwise always as a document.

progress_callback (`callable`, optional): A callback function accepting two parameters: `(sent bytes, total)`.

reply_to (`int` | Message): Same as reply_to from .send_message().

attributes (`list`, optional): Optional attributes that override the inferred ones, like DocumentAttributeFilename and so on.

thumb (`str` | `bytes` | `file`, optional): Optional thumbnail (for videos).

allow_cache (`bool`, optional): Whether to allow using the cached version stored in the database or not. Defaults to `True` to avoid re-uploads. Must be `False` if you wish to use different attributes or thumb than those that were used when the file was cached.

parse_mode (`str`, optional): The parse mode for the caption message.

Kwargs: If "is_voice_note" in kwargs, despite its value, and the file is sent as a document, it will be sent as a voice note.

Notes: If the `hachoir3` package (`hachoir` module) is installed, it will be used to determine metadata from audio and video files.

Returns: The Message (or messages) containing the sent file, or messages if a list of them was passed.

`send_message`(*entity*, *message=''*, *reply_to=None*, *parse_mode='md'*, *link_preview=True*, *file=None*, *force_document=False*, *clear_draft=False*)

Sends the given message to the specified entity (user/chat/channel).

The default parse mode is the same as the official applications (a custom flavour of markdown). `**bold**`, `` `code` `` or `__italic__` are available. In addition you can send `[links](https://example.com)` and `[mentions](@username)` (or using IDs like in the Bot API: `[mention](tg://user?id=123456789)`) and `pre` blocks with three backticks.

Args:

entity (`entity`): To who will it be sent.

message (`str` | Message): The message to be sent, or another message object to resend.

reply_to (`int` | Message, optional): Whether to reply to a message or not. If an integer is provided, it should be the ID of the message that it should reply to.

parse_mode (`str`, optional): Can be 'md' or 'markdown' for markdown-like parsing (default), or 'htm' or 'html' for HTML-like parsing. If `None` or any other false-y value is provided, the message will be sent with no formatting.

link_preview (`bool`, optional): Should the link preview be shown?

file (`file`, optional): Sends a message with a file attached (e.g. a photo, video, audio or document). The `message` may be empty.

force_document (`bool`, optional): Whether to send the given file as a document or not.

clear_draft (`bool`, optional): Whether the existing draft should be cleared or not. Has no effect when sending a file.

Returns: The sent Message.

`send_read_acknowledge`(*entity*, *message=None*, *max_id=None*, *clear_mentions=False*)

Sends a "read acknowledge" (i.e., notifying the given peer that we've read their messages, also known as the "double check").

Args:

entity (`entity`): The chat where these messages are located.

message (list | Message): Either a list of messages or a single message.

max_id (int): Overrides messages, until which message should the acknowledge should be sent.

clear_mentions (bool): Whether the mention badge should be cleared (so that there are no more mentions) or not for the given entity.

If no message is provided, this will be the only action taken.

send_voice_note (**args*, ***kwargs*)
Wrapper method around *send_file()* with is_voice_note=True.

sign_in (*phone=None*, *code=None*, *password=None*, *bot_token=None*, *phone_code_hash=None*)
Starts or completes the sign in process with the given phone number or code that Telegram sent.

Args:

phone (str | int): The phone to send the code to if no code was provided, or to override the phone that was previously used with these requests.

code (str | int): The code that Telegram sent. Note that if you have sent this code through the application itself it will immediately expire. If you want to send the code, obfuscate it somehow. If you're not doing any of this you can ignore this note.

password (str): 2FA password, should be used if a previous call raised SessionPasswordNeededError.

bot_token (str): Used to sign in as a bot. Not all requests will be available. This should be the hash the @BotFather gave you.

phone_code_hash (str): The hash returned by .send_code_request. This can be set to None to use the last hash known.

Returns: The signed in user, or the information about *send_code_request()*.

sign_up (*code*, *first_name*, *last_name=''*)
Signs up to Telegram if you don't have an account yet. You must call .send_code_request(phone) first.

Args:

code (str | int): The code sent by Telegram

first_name (str): The first name to be used by the new account.

last_name (str, optional) Optional last name.

Returns: The new created User.

start (*phone=<function TelegramClient.<lambda>>*, *password=<function TelegramClient.<lambda>>*, *bot_token=None*, *force_sms=False*, *code_callback=None*, *first_name='New User'*, *last_name=''*)
Convenience method to interactively connect and sign in if required, also taking into consideration that 2FA may be enabled in the account.

Example usage:

```
>>> client = TelegramClient(session, api_id, api_hash).start(phone)
Please enter the code you received: 12345
Please enter your password: *******
(You are now logged in)
```

Args:

phone (str | int | callable): The phone (or callable without arguments to get it) to which the code will be sent.

password (`callable`, optional): The password for 2 Factor Authentication (2FA). This is only required if it is enabled in your account.

bot_token (`str`): Bot Token obtained by @BotFather to log in as a bot. Cannot be specified with `phone` (only one of either allowed).

force_sms (`bool`, optional): Whether to force sending the code request as SMS. This only makes sense when signing in with a `phone`.

code_callback (`callable`, optional): A callable that will be used to retrieve the Telegram login code. Defaults to `input()`.

first_name (`str`, optional): The first name to be used if signing up. This has no effect if the account already exists and you sign in.

last_name (`str`, optional): Similar to the first name, but for the last. Optional.

Returns: This *`TelegramClient`*, so initialization can be chained with `.start()`.

`upload_file`(*file*, *part_size_kb=None*, *file_name=None*, *use_cache=None*, *progress_callback=None*)

Uploads the specified file and returns a handle (an instance of InputFile or InputFileBig, as required) which can be later used before it expires (they are usable during less than a day).

Uploading a file will simply return a "handle" to the file stored remotely in the Telegram servers, which can be later used on. This will **not** upload the file to your own chat or any chat at all.

Args:

file (`str | bytes | file`): The path of the file, byte array, or stream that will be sent. Note that if a byte array or a stream is given, a filename or its type won't be inferred, and it will be sent as an "unnamed application/octet-stream".

Subsequent calls with the very same file will result in immediate uploads, unless `.clear_file_cache()` is called.

part_size_kb (`int`, optional): Chunk size when uploading files. The larger, the less requests will be made (up to 512KB maximum).

file_name (`str`, optional): The file name which will be used on the resulting InputFile. If not specified, the name will be taken from the `file` and if this is not a `str`, it will be `"unnamed"`.

use_cache (`type`, optional): The type of cache to use (currently either `InputDocument` or `InputPhoto`). If present and the file is small enough to need the MD5, it will be checked against the database, and if a match is found, the upload won't be made. Instead, an instance of type `use_cache` will be returned.

progress_callback (`callable`, optional): A callback function accepting two parameters: `(sent bytes, total)`.

Returns: InputFileBig if the file size is larger than 10MB, `InputSizedFile` (subclass of InputFile) otherwise.

telethon.telegram_bare_client module

class telethon.telegram_bare_client.**TelegramBareClient**(*session*, *api_id*, *api_hash*, *connection_mode=<ConnectionMode.TCP_FULL: 1>*, *use_ipv6=False*, *proxy=None*, *update_workers=None*, *spawn_read_thread=False*, *timeout=datetime.timedelta(0, 5)*, *report_errors=True*, *device_model=None*, *system_version=None*, *app_version=None*, *lang_code='en'*, *system_lang_code='en'*)

Bases: `object`

Bare Telegram Client with just the minimum -

The reason to distinguish between a MtProtoSender and a TelegramClient itself is because the sender is just that, a sender, which should know nothing about Telegram but rather how to handle this specific connection.

The TelegramClient itself should know how to initialize a proper connection to the servers, as well as other basic methods such as disconnection and reconnection.

This distinction between a bare client and a full client makes it possible to create clones of the bare version (by using the same session, IP address and port) to be able to execute queries on either, without the additional cost that would involve having the methods for signing in, logging out, and such.

connect(*_sync_updates=True*)

Connects to the Telegram servers, executing authentication if required. Note that authenticating to the Telegram servers is not the same as authenticating the desired user itself, which may require a call (or several) to 'sign_in' for the first time.

Note that the optional parameters are meant for internal use.

If '_sync_updates', sync_updates() will be called and a second thread will be started if necessary. Note that this will FAIL if the client is not connected to the user's native data center, raising a "UserMigrateError", and calling .disconnect() in the process.

disconnect()

Disconnects from the Telegram server and stops all the spawned threads

get_input_entity(*peer*)

Stub method, no functionality so that calling `.get_input_entity()` from `.resolve()` doesn't fail.

idle(*stop_signals=(<Signals.SIGINT: 2>, <Signals.SIGTERM: 15>, <Signals.SIGIOT: 6>)*)

Idles the program by looping forever and listening for updates until one of the signals are received, which breaks the loop.

Parameters **stop_signals** – Iterable containing signals from the signal module that will be subscribed to TelegramClient.disconnect() (effectively stopping the idle loop), which will be called on receiving one of those signals.

Returns

invoke(**requests*, *retries=5*)

Invokes (sends) a MTProtoRequest and returns (receives) its result.

The invoke will be retried up to 'retries' times before raising RuntimeError().

is_connected()

is_user_authorized()
Has the user been authorized yet (code request sent and confirmed)?

set_proxy(*proxy*)
Change the proxy used by the connections.

sync_updates()
Synchronizes self.updates to their initial state. Will be called automatically on connection if self.updates.enabled = True, otherwise it should be called manually after enabling updates.

telethon.utils module

Utilities for working with the Telegram API itself (such as handy methods to convert between an entity like an User, Chat, etc. into its Input version)

telethon.utils.**get_appropriated_part_size**(*file_size*)
Gets the appropriated part size when uploading or downloading files, given an initial file size.

telethon.utils.**get_display_name**(*entity*)
Gets the display name for the given entity, if it's an User, Chat or Channel. Returns an empty string otherwise.

telethon.utils.**get_extension**(*media*)
Gets the corresponding extension for any Telegram media.

telethon.utils.**get_input_channel**(*entity*)
Similar to *get_input_peer()*, but for InputChannel's alone.

telethon.utils.**get_input_document**(*document*)
Similar to *get_input_peer()*, but for documents

telethon.utils.**get_input_geo**(*geo*)
Similar to *get_input_peer()*, but for geo points

telethon.utils.**get_input_media**(*media*, *is_photo=False*)
Similar to *get_input_peer()*, but for media.

If the media is a file location and `is_photo` is known to be `True`, it will be treated as an InputMediaUploadedPhoto.

telethon.utils.**get_input_peer**(*entity*, *allow_self=True*)
Gets the input peer for the given "entity" (user, chat or channel). A `TypeError` is raised if the given entity isn't a supported type.

telethon.utils.**get_input_photo**(*photo*)
Similar to *get_input_peer()*, but for photos

telethon.utils.**get_input_user**(*entity*)
Similar to *get_input_peer()*, but for InputUser's alone.

telethon.utils.**get_peer_id**(*peer*)
Finds the ID of the given peer, and converts it to the "bot api" format so it the peer can be identified back. User ID is left unmodified, chat ID is negated, and channel ID is prefixed with -100.

The original ID and the peer type class can be returned with a call to `resolve_id(marked_id)()`.

telethon.utils.**is_audio**(*file*)
Returns `True` if the file extension looks like an audio file.

telethon.utils.**is_image**(*file*)
Returns `True` if the file extension looks like an image file to Telegram.

telethon.utils.**is_list_like**(*obj*)
Returns `True` if the given object looks like a list.

Checking `if hasattr(obj, '__iter__')` and ignoring `str/bytes` is not enough. Things like `open()` are also iterable (and probably many other things), so just support the commonly known list-like objects.

telethon.utils.**is_video**(*file*)
Returns `True` if the file extension looks like a video file.

telethon.utils.**parse_phone**(*phone*)
Parses the given phone, or returns `None` if it's invalid.

telethon.utils.**parse_username**(*username*)
Parses the given username or channel access hash, given a string, username or URL. Returns a tuple consisting of both the stripped, lowercase username and whether it is a joinchat/ hash (in which case is not lowercase'd).

Returns `None` if the `username` is not valid.

telethon.utils.**resolve_id**(*marked_id*)
Given a marked ID, returns the original ID and its Peer type.

telethon.helpers module

Various helpers not related to the Telegram API itself

telethon.helpers.**calc_key**(*auth_key*, *msg_key*, *client*)
Calculate the key based on Telegram guidelines for MtProto 2, specifying whether it's the client or not.

telethon.helpers.**ensure_parent_dir_exists**(*file_path*)
Ensures that the parent directory exists

telethon.helpers.**generate_key_data_from_nonce**(*server_nonce*, *new_nonce*)
Generates the key data corresponding to the given nonce

telethon.helpers.**generate_random_long**(*signed=True*)
Generates a random long integer (8 bytes), which is optionally signed

telethon.helpers.**get_password_hash**(*pw*, *current_salt*)
Gets the password hash for the two-step verification. current_salt should be the byte array provided by invoking GetPasswordRequest()

telethon.helpers.**pack_message**(*session*, *message*)
Packs a message following MtProto 2.0 guidelines

telethon.helpers.**unpack_message**(*session*, *reader*)
Unpacks a message following MtProto 2.0 guidelines

telethon.events package

telethon.events package

exception telethon.events.**StopPropagation**
Bases: `Exception`

If this exception is raised in any of the handlers for a given event, it will stop the execution of all other registered event handlers. It can be seen as the `StopIteration` in a for loop but for events.

Example usage:

```
>>> from telethon import TelegramClient, events
>>> client = TelegramClient(...)
>>>
>>> @client.on(events.NewMessage)
... def delete(event):
...     event.delete()
...     # No other event handler will have a chance to handle this event
...     raise StopPropagation
...
>>> @client.on(events.NewMessage)
... def _(event):
...     # Will never be reached, because it is the second handler
...     pass
```

Every event (builder) subclasses *telethon.events.common.EventBuilder*, so all the methods in it can be used from any event builder/event instance.

class telethon.events.common.**EventBuilder**(*chats=None*, *blacklist_chats=False*)

Bases: abc.ABC

The common event builder, with builtin support to filter per chat.

Args:

chats (entity, optional): May be one or more entities (username/peer/etc.). By default, only matching chats will be handled.

blacklist_chats (bool, optional): Whether to treat the chats as a blacklist instead of as a whitelist (default). This means that every chat will be handled *except* those specified in `chats` which will be ignored if `blacklist_chats=True`.

build(*update*)

Builds an event for the given update if possible, or returns None

resolve(*client*)

Helper method to allow event builders to be resolved before usage

class telethon.events.common.**EventCommon**(*chat_peer=None*, *msg_id=None*, *broadcast=False*)

Bases: abc.ABC

Intermediate class with common things to all events

chat

The (User | Chat | Channel, optional) on which the event occurred. This property may make an API call the first time to get the most up to date version of the chat (mostly when the event doesn't belong to a channel), so keep that in mind.

client

input_chat

The (InputPeer) (group, megagroup or channel) on which the event occurred. This doesn't have the title or anything, but is useful if you don't need those to avoid further requests.

Note that this might be `None` if the library can't find it.

stringify()

to_dict()

class telethon.events.common.**Raw**(*chats=None*, *blacklist_chats=False*)
Bases: *telethon.events.common.EventBuilder*

Represents a raw event. The event is the update itself.

build(*update*)

resolve(*client*)

telethon.events.common.**name_inner_event**(*cls*)
Decorator to rename cls.Event 'Event' as 'cls.Event'

Below all the event types are listed:

class telethon.events.newmessage.**NewMessage**(*incoming=None*, *outgoing=None*, *chats=None*, *blacklist_chats=False*, *pattern=None*)
Bases: *telethon.events.common.EventBuilder*

Represents a new message event builder.

Args:

incoming (bool, optional): If set to `True`, only **incoming** messages will be handled. Mutually exclusive with `outgoing` (can only set one of either).

outgoing (bool, optional): If set to `True`, only **outgoing** messages will be handled. Mutually exclusive with `incoming` (can only set one of either).

pattern (str, callable, Pattern, optional): If set, only messages matching this pattern will be handled. You can specify a regex-like string which will be matched against the message, a callable function that returns `True` if a message is acceptable, or a compiled regex pattern.

class Event(*message*)
Bases: *telethon.events.common.EventCommon*

Represents the event of a new message.

Members:

message (Message): This is the original Message object.

is_private (bool): True if the message was sent as a private message.

is_group (bool): True if the message was sent on a group or megagroup.

is_channel (bool): True if the message was sent on a megagroup or channel.

is_reply (str): Whether the message is a reply to some other or not.

audio
If the message media is a document with an Audio attribute, this returns the Document object.

delete(**args*, ***kwargs*)
Deletes the message. You're responsible for checking whether you have the permission to do so, or to except the error otherwise. This is a shorthand for `client.delete_messages(event.chat, event.message, ...)`.

document
If the message media is a document, this returns the Document object.

edit(**args*, ***kwargs*)
Edits the message iff it's outgoing. This is a shorthand for `client.edit_message(event.chat, event.message, ...)`.

Returns `None` if the message was incoming, or the edited message otherwise.

forward
The unmodified MessageFwdHeader, if present..

forward_to(**args*, ***kwargs*)
Forwards the message. This is a shorthand for `client.forward_messages(entity, event.message, event.chat)`.

gif
If the message media is a document with an Animated attribute, this returns the Document object.

input_sender
This (InputPeer) is the input version of the user who sent the message. Similarly to `input_chat`, this doesn't have things like username or similar, but still useful in some cases.

Note that this might not be available if the library can't find the input chat, or if the message a broadcast on a channel.

media
The unmodified MessageMedia, if present.

out
Whether the message is outgoing (i.e. you sent it from another session) or incoming (i.e. someone else sent it).

photo
If the message media is a photo, this returns the Photo object.

raw_text
The raw message text, ignoring any formatting.

reply(**args*, ***kwargs*)
Replies to the message (as a reply). This is a shorthand for `client.send_message(event.chat, ..., reply_to=event.message.id)`.

reply_message
This optional Message will make an API call the first time to get the full Message object that one was replying to, so use with care as there is no caching besides local caching yet.

respond(**args*, ***kwargs*)
Responds to the message (not as a reply). This is a shorthand for `client.send_message(event.chat, ...)`.

sender
This (User) may make an API call the first time to get the most up to date version of the sender (mostly when the event doesn't belong to a channel), so keep that in mind.

`input_sender` needs to be available (often the case).

sticker
If the message media is a document with a Sticker attribute, this returns the Document object.

text
The message text, markdown-formatted.

video
If the message media is a document with a Video attribute, this returns the Document object.

video_note
If the message media is a document with a Video attribute, this returns the Document object.

voice
If the message media is a document with a Voice attribute, this returns the Document object.

build(*update*)

class telethon.events.chataction.**ChatAction**(*chats=None*, *blacklist_chats=False*)
Bases: *telethon.events.common.EventBuilder*

Represents an action in a chat (such as user joined, left, or new pin).

class Event(*where*, *new_pin=None*, *new_photo=None*, *added_by=None*, *kicked_by=None*, *created=None*, *users=None*, *new_title=None*, *unpin=None*)
Bases: *telethon.events.common.EventCommon*

Represents the event of a new chat action.

Members:

action_message (MessageAction): The message invoked by this Chat Action.

new_pin (bool): `True` if there is a new pin.

new_photo (bool): `True` if there's a new chat photo (or it was removed).

photo (Photo, optional): The new photo (or `None` if it was removed).

user_added (bool): `True` if the user was added by some other.

user_joined (bool): `True` if the user joined on their own.

user_left (bool): `True` if the user left on their own.

user_kicked (bool): `True` if the user was kicked by some other.

created (bool, optional): `True` if this chat was just created.

new_title (bool, optional): The new title string for the chat, if applicable.

unpin (bool): `True` if the existing pin gets unpinned.

added_by
The user who added `users`, if applicable (`None` otherwise).

delete(**args*, ***kwargs*)
Deletes the chat action message. You're responsible for checking whether you have the permission to do so, or to except the error otherwise. This is a shorthand for `client.delete_messages(event.chat, event.message, ...)`.

Does nothing if no message action triggered this event.

input_user
Input version of the `self.user` property.

input_users
Input version of the `self.users` property.

kicked_by
The user who kicked `users`, if applicable (`None` otherwise).

pinned_message
If `new_pin` is `True`, this returns the (Message) object that was pinned.

reply(**args*, ***kwargs*)
Replies to the chat action message (as a reply). Shorthand for `client.send_message(event.chat, ..., reply_to=event.message.id)`.

Has the same effect as `.respond()` if there is no message.

respond(**args*, ***kwargs*)
Responds to the chat action message (not as a reply). Shorthand for `client.send_message(event.chat, ...)`.

user
The single user that takes part in this action (e.g. joined).

Might be `None` if the information can't be retrieved or there is no user taking part.

users
A list of users that take part in this action (e.g. joined).

Might be empty if the information can't be retrieved or there are no users taking part.

build(*update*)

class telethon.events.userupdate.**UserUpdate**(*chats=None*, *blacklist_chats=False*)
Bases: *telethon.events.common.EventBuilder*

Represents an user update (gone online, offline, joined Telegram).

class Event(*user_id*, *status=None*, *typing=None*)
Bases: *telethon.events.common.EventCommon*

Represents the event of an user status update (last seen, joined).

Members:

online (bool, optional): `True` if the user is currently online, `False` otherwise. Might be `None` if this information is not present.

last_seen (datetime, optional): Exact date when the user was last seen if known.

until (datetime, optional): Until when will the user remain online.

within_months (bool): `True` if the user was seen within 30 days.

within_weeks (bool): `True` if the user was seen within 7 days.

recently (bool): `True` if the user was seen within a day.

action (SendMessageAction, optional): The "typing" action if any the user is performing if any.

cancel (bool): `True` if the action was cancelling other actions.

typing (bool): `True` if the action is typing a message.

recording (bool): `True` if the action is recording something.

uploading (bool): `True` if the action is uploading something.

playing (bool): `True` if the action is playing a game.

audio (bool): `True` if what's being recorded/uploaded is an audio.

round (bool): `True` if what's being recorded/uploaded is a round video.

video (bool): `True` if what's being recorded/uploaded is an video.

document (bool): `True` if what's being uploaded is document.

geo (bool): `True` if what's being uploaded is a geo.

photo (bool): `True` if what's being uploaded is a photo.

contact (bool): `True` if what's being uploaded (selected) is a contact.

user
Alias around the chat (conversation).

build(*update*)

class telethon.events.messageedited.**MessageEdited**(*incoming=None*, *outgoing=None*, *chats=None*, *blacklist_chats=False*, *pattern=None*)

Bases: *telethon.events.newmessage.NewMessage*

Event fired when a message has been edited.

class Event(*message*)

Bases: telethon.events.newmessage.Event

build(*update*)

class telethon.events.messagedeleted.**MessageDeleted**(*chats=None*, *blacklist_chats=False*)

Bases: *telethon.events.common.EventBuilder*

Event fired when one or more messages are deleted.

class Event(*deleted_ids*, *peer*)

Bases: *telethon.events.common.EventCommon*

build(*update*)

class telethon.events.messageread.**MessageRead**(*inbox=False*, *chats=None*, *blacklist_chats=None*)

Bases: *telethon.events.common.EventBuilder*

Event fired when one or more messages have been read.

Args:

inbox (bool, optional): If this argument is `True`, then when you read someone else's messages the event will be fired. By default (`False`) only when messages you sent are read by someone else will fire it.

class Event(*peer=None*, *max_id=None*, *out=False*, *contents=False*, *message_ids=None*)

Bases: *telethon.events.common.EventCommon*

Represents the event of one or more messages being read.

Members:

max_id (int): Up to which message ID has been read. Every message with an ID equal or lower to it have been read.

outbox (bool): `True` if someone else has read your messages.

contents (bool): `True` if what was read were the contents of a message. This will be the case when e.g. you play a voice note. It may only be set on `inbox` events.

inbox

`True` if you have read someone else's messages.

is_read(*message*)

Returns `True` if the given message (or its ID) has been read.

If a list-like argument is provided, this method will return a list of booleans indicating which messages have been read.

message_ids

The IDs of the messages **which contents'** were read.

Use *is_read()* if you need to check whether a message was read instead checking if it's in here.

messages
The list of Message **which contents'** were read.

Use *is_read()* if you need to check whether a message was read instead checking if it's in here.

build(*update*)

telethon.update_state module

class telethon.update_state.**UpdateState**(*workers=None*)
Bases: object

Used to hold the current state of processed updates. To retrieve an update, *poll()* should be called.

WORKER_POLL_TIMEOUT = 5.0

can_poll()
Returns True if a call to .poll() won't lock

get_workers()

poll(*timeout=None*)
Polls an update or blocks until an update object is available. If 'timeout is not None', it should be a floating point value, and the method will 'return None' if waiting times out.

process(*update*)
Processes an update object. This method is normally called by the library itself.

set_workers(*n*)
Changes the number of workers running. If 'n is None', clears all pending updates from memory.

setup_workers()

stop_workers()
Waits for all the worker threads to stop.

workers

telethon.sessions module

telethon.cryto package

telethon.crypto package

telethon.crypto.aes module

AES IGE implementation in Python. This module may use libssl if available.

class telethon.crypto.aes.**AES**
Bases: object

Class that servers as an interface to encrypt and decrypt text through the AES IGE mode.

static decrypt_ige(*cipher_text*, *key*, *iv*)
Decrypts the given text in 16-bytes blocks by using the given key and 32-bytes initialization vector.

static encrypt_ige(*plain_text*, *key*, *iv*)
Encrypts the given text in 16-bytes blocks by using the given key and 32-bytes initialization vector.

telethon.crypto.aes_ctr module

This module holds the AESModeCTR wrapper class.

class telethon.crypto.aes_ctr.**AESModeCTR**(*key*, *iv*)
Bases: object

Wrapper around pyaes.AESModeOfOperationCTR mode with custom IV

decrypt(*data*)
Decrypts the given cipher text through AES CTR

Parameters **data** – the cipher text to be decrypted.

Returns the decrypted plain text.

encrypt(*data*)
Encrypts the given plain text through AES CTR.

Parameters **data** – the plain text to be encrypted.

Returns the encrypted cipher text.

telethon.crypto.auth_key module

This module holds the AuthKey class.

class telethon.crypto.auth_key.**AuthKey**(*data*)
Bases: object

Represents an authorization key, used to encrypt and decrypt messages sent to Telegram's data centers.

calc_new_nonce_hash(*new_nonce*, *number*)
Calculates the new nonce hash based on the current attributes.

Parameters

- **new_nonce** – the new nonce to be hashed.
- **number** – number to prepend before the hash.

Returns the hash for the given new nonce.

telethon.crypto.cdn_decrypter module

This module holds the CdnDecrypter utility class.

class telethon.crypto.cdn_decrypter.**CdnDecrypter**(*cdn_client*, *file_token*, *cdn_aes*, *cdn_file_hashes*)
Bases: object

Used when downloading a file results in a 'FileCdnRedirect' to both prepare the redirect, decrypt the file as it downloads, and ensure the file hasn't been tampered. https://core.telegram.org/cdn

static check(*data*, *cdn_hash*)
Checks the integrity of the given data. Raises CdnFileTamperedError if the integrity check fails.

Parameters

- **data** – the data to be hashed.
- **cdn_hash** – the expected hash.

get_file()
Calls GetCdnFileRequest and decrypts its bytes. Also ensures that the file hasn't been tampered.

Returns the CdnFile result.

static prepare_decrypter(*client*, *cdn_client*, *cdn_redirect*)
Prepares a new CDN decrypter.

Parameters

- **client** – a TelegramClient connected to the main servers.
- **cdn_client** – a new client connected to the CDN.
- **cdn_redirect** – the redirect file object that caused this call.

Returns (CdnDecrypter, first chunk file data)

telethon.crypto.factorization module

This module holds a fast Factorization class.

class telethon.crypto.factorization.**Factorization**
Bases: object

Simple module to factorize large numbers really quickly.

classmethod factorize(*pq*)
Factorizes the given large integer.

Parameters **pq** – the prime pair pq.

Returns a tuple containing the two factors p and q.

static gcd(*a*, *b*)
Calculates the Greatest Common Divisor.

Parameters

- **a** – the first number.
- **b** – the second number.

Returns GCD(a, b)

telethon.crypto.rsa module

This module holds several utilities regarding RSA and server fingerprints.

telethon.crypto.rsa.**add_key**(*pub*)
Adds a new public key to be used when encrypting new data is needed

telethon.crypto.rsa.**encrypt**(*fingerprint*, *data*)
Encrypts the given data known the fingerprint to be used in the way Telegram requires us to do so (sha1(data) + data + padding)

Parameters

- **fingerprint** – the fingerprint of the RSA key.
- **data** – the data to be encrypted.

Returns the cipher text, or None if no key matching this fingerprint is found.

telethon.crypto.rsa.**get_byte_array**(*integer*)
Return the variable length bytes corresponding to the given int

telethon.errors package

telethon.errors package

telethon.errors.common module

Errors not related to the Telegram API itself

exception telethon.errors.common.**BrokenAuthKeyError**
Bases: Exception

Occurs when the authorization key for a data center is not valid.

exception telethon.errors.common.**CdnFileTamperedError**
Bases: *telethon.errors.common.SecurityError*

Occurs when there's a hash mismatch between the decrypted CDN file and its expected hash.

exception telethon.errors.common.**InvalidChecksumError**(*checksum*, *valid_checksum*)
Bases: Exception

Occurs when using the TCP full mode and the checksum of a received packet doesn't match the expected checksum.

exception telethon.errors.common.**ReadCancelledError**
Bases: Exception

Occurs when a read operation was cancelled.

exception telethon.errors.common.**SecurityError**(**args*)
Bases: Exception

Generic security error, mostly used when generating a new AuthKey.

exception telethon.errors.common.**TypeNotFoundError**(*invalid_constructor_id*)
Bases: Exception

Occurs when a type is not found, for example, when trying to read a TLObject with an invalid constructor code.

telethon.errors.rpc_base_errors module

exception telethon.errors.rpc_base_errors.**AuthKeyError**(*message*)
Bases: *telethon.errors.rpc_base_errors.RPCError*

Errors related to invalid authorization key, like AUTH_KEY_DUPLICATED which can cause the connection to fail.

code = 406

message = 'AUTH_KEY'

exception telethon.errors.rpc_base_errors.**BadMessageError**(*code*)
Bases: Exception

Occurs when handling a bad_message_notification.

ErrorMessages = {16: 'msg_id too low (most likely, client time is wrong it would be w

exception telethon.errors.rpc_base_errors.**BadRequestError**
Bases: *telethon.errors.rpc_base_errors.RPCError*

The query contains errors. In the event that a request was created using a form and contains user generated data, the user should be notified that the data must be corrected before the query is repeated.

code = 400

message = 'BAD_REQUEST'

exception telethon.errors.rpc_base_errors.**FloodError**
Bases: *telethon.errors.rpc_base_errors.RPCError*

The maximum allowed number of attempts to invoke the given method with the given input parameters has been exceeded. For example, in an attempt to request a large number of text messages (SMS) for the same phone number.

code = 420

message = 'FLOOD'

exception telethon.errors.rpc_base_errors.**ForbiddenError**(*message*)
Bases: *telethon.errors.rpc_base_errors.RPCError*

Privacy violation. For example, an attempt to write a message to someone who has blacklisted the current user.

code = 403

message = 'FORBIDDEN'

exception telethon.errors.rpc_base_errors.**InvalidDCError**
Bases: *telethon.errors.rpc_base_errors.RPCError*

The request must be repeated, but directed to a different data center.

code = 303

message = 'ERROR_SEE_OTHER'

exception telethon.errors.rpc_base_errors.**NotFoundError**(*message*)
Bases: *telethon.errors.rpc_base_errors.RPCError*

An attempt to invoke a non-existent object, such as a method.

code = 404

message = 'NOT_FOUND'

exception telethon.errors.rpc_base_errors.**RPCError**
Bases: Exception

Base class for all Remote Procedure Call errors.

code = None

message = None

exception telethon.errors.rpc_base_errors.**ServerError**(*message*)
Bases: *telethon.errors.rpc_base_errors.RPCError*

An internal server error occurred while a request was being processed for example, there was a disruption while accessing a database or file storage.

code = 500

message = 'INTERNAL'

exception telethon.errors.rpc_base_errors.**UnauthorizedError**
Bases: *telethon.errors.rpc_base_errors.RPCError*

There was an unauthorized attempt to use functionality available only to authorized users.

code = 401

message = 'UNAUTHORIZED'

telethon.extensions package

telethon.extensions package

telethon.extensions.binary_reader module

This module contains the BinaryReader utility class.

class telethon.extensions.binary_reader.**BinaryReader**(*data=None*, *stream=None*)
Bases: object

Small utility class to read binary data. Also creates a "Memory Stream" if necessary

close()
Closes the reader, freeing the BytesIO stream.

get_bytes()
Gets the byte array representing the current buffer as a whole.

read(*length=None*)
Read the given amount of bytes.

read_byte()
Reads a single byte value.

read_double()
Reads a real floating point (8 bytes) value.

read_float()
Reads a real floating point (4 bytes) value.

read_int(*signed=True*)
Reads an integer (4 bytes) value.

read_large_int(*bits*, *signed=True*)
Reads a n-bits long integer value.

read_long(*signed=True*)
Reads a long integer (8 bytes) value.

seek(*offset*)
Seeks the stream position given an offset from the current position. The offset may be negative.

set_position(*position*)
Sets the current position on the stream.

tell_position()
Tells the current position on the stream.

tgread_bool()
Reads a Telegram boolean value.

tgread_bytes()
Reads a Telegram-encoded byte array, without the need of specifying its length.

tgread_date()
Reads and converts Unix time (used by Telegram) into a Python datetime object.

tgread_object()
Reads a Telegram object.

tgread_string()
Reads a Telegram-encoded string.

tgread_vector()
Reads a vector (a list) of Telegram objects.

telethon.extensions.markdown module

Simple markdown parser which does not support nesting. Intended primarily for use within the library, which attempts to handle emojies correctly, since they seem to count as two characters and it's a bit strange.

telethon.extensions.markdown.**get_inner_text**(*text*, *entity*)
Gets the inner text that's surrounded by the given entity or entities. For instance: text = 'hey!', entity = MessageEntityBold(2, 2) -> 'y!'.

Parameters

- **text** – the original text.
- **entity** – the entity or entities that must be matched.

Returns a single result or a list of the text surrounded by the entities.

telethon.extensions.markdown.**parse**(*message*, *delimiters=None*, *url_re=None*)
Parses the given markdown message and returns its stripped representation plus a list of the MessageEntity's that were found.

Parameters

- **message** – the message with markdown-like syntax to be parsed.
- **delimiters** – the delimiters to be used, {delimiter: type}.
- **url_re** – the URL bytes regex to be used. Must have two groups.

Returns a tuple consisting of (clean message, [message entities]).

telethon.extensions.markdown.**unparse**(*text*, *entities*, *delimiters=None*, *url_fmt=None*)
Performs the reverse operation to .parse(), effectively returning markdown-like syntax given a normal text and its MessageEntity's.

Parameters

- **text** – the text to be reconverted into markdown.
- **entities** – the MessageEntity's applied to the text.

Returns a markdown-like text representing the combination of both inputs.

telethon.extensions.tcp_client module

This module holds a rough implementation of the C# TCP client.

class telethon.extensions.tcp_client.**TcpClient**(*proxy=None*, *timeout=datetime.timedelta(0, 5)*)

Bases: object

A simple TCP client to ease the work with sockets and proxies.

close()
Closes the connection.

connect(*ip*, *port*)
Tries connecting forever to IP:port unless an OSError is raised.

Parameters

- **ip** – the IP to connect to.
- **port** – the port to connect to.

connected
Determines whether the client is connected or not.

read(*size*)
Reads (receives) a whole block of size bytes from the connected peer.

Parameters **size** – the size of the block to be read.

Returns the read data with len(data) == size.

write(*data*)
Writes (sends) the specified bytes to the connected peer.

Parameters **data** – the data to send.

telethon.network package

telethon.network package

telethon.network.authenticator module

This module contains several functions that authenticate the client machine with Telegram's servers, effectively creating an authorization key.

telethon.network.authenticator.**do_authentication**(*connection*, *retries=5*)
Performs the authentication steps on the given connection. Raises an error if all attempts fail.

Parameters

- **connection** – the connection to be used (must be connected).
- **retries** – how many times should we retry on failure.

Returns

telethon.network.authenticator.**get_int**(*byte_array*, *signed=True*)
Gets the specified integer from its byte array. This should be used by this module alone, as it works with big endian.

Parameters

- **byte_array** – the byte array representing th integer.
- **signed** – whether the number is signed or not.

Returns the integer representing the given byte array.

telethon.network.connection module

This module holds both the Connection class and the ConnectionMode enum, which specifies the protocol to be used by the Connection.

class telethon.network.connection.**Connection**(*mode=<ConnectionMode.TCP_FULL: 1>*, *proxy=None*, *timeout=datetime.timedelta(0, 5)*)

Bases: object

Represents an abstract connection (TCP, TCP abridged...). 'mode' must be any of the ConnectionMode enumeration.

Note that '.send()' and '.recv()' refer to messages, which will be packed accordingly, whereas '.write()' and '.read()' work on plain bytes, with no further additions.

clone()
Creates a copy of this Connection.

close()
Closes the connection.

connect(*ip*, *port*)
Estabilishes a connection to IP:port.

Parameters

- **ip** – the IP to connect to.
- **port** – the port to connect to.

get_timeout()
Returns the timeout used by the connection.

is_connected()
Determines whether the connection is alive or not.

Returns true if it's connected.

read(*length*)

recv()
Receives and unpacks a message

send(*message*)
Encapsulates and sends the given message

write(*data*)

class telethon.network.connection.**ConnectionMode**

Bases: enum.Enum

Represents which mode should be used to stabilise a connection.

TCP_FULL: Default Telegram mode. Sends 12 additional bytes and needs to calculate the CRC value of the packet itself.

TCP_INTERMEDIATE: Intermediate mode between TCP_FULL and TCP_ABRIDGED. Always sends 4 extra bytes for the packet length.

TCP_ABRIDGED: This is the mode with the lowest overhead, as it will only require 1 byte if the packet length is less than 508 bytes (127 << 2, which is very common).

TCP_OBFUSCATED: Encodes the packet just like TCP_ABRIDGED, but encrypts every message with a randomly generated key using the AES-CTR mode so the packets are harder to discern.

TCP_ABRIDGED = 3

TCP_FULL = 1

TCP_INTERMEDIATE = 2

TCP_OBFUSCATED = 4

telethon.network.mtproto_plain_sender module

This module contains the class used to communicate with Telegram's servers in plain text, when no authorization key has been created yet.

class telethon.network.mtproto_plain_sender.**MtProtoPlainSender**(*connection*)
Bases: object

MTProto Mobile Protocol plain sender (https://core.telegram.org/mtproto/description#unencrypted-messages)

connect()
Connects to Telegram's servers.

disconnect()
Disconnects from Telegram's servers.

receive()
Receives a plain packet from the network.

Returns the response body.

send(*data*)
Sends a plain packet (auth_key_id = 0) containing the given message body (data).

Parameters **data** – the data to be sent.

telethon.network.mtproto_sender module

This module contains the class used to communicate with Telegram's servers encrypting every packet, and relies on a valid AuthKey in the used Session.

class telethon.network.mtproto_sender.**MtProtoSender**(*session*, *connection*)
Bases: object

MTProto Mobile Protocol sender (https://core.telegram.org/mtproto/description).

Note that this class is not thread-safe, and calling send/receive from two or more threads at the same time is undefined behaviour. Rationale:

a new connection should be spawned to send/receive requests in parallel, so thread-safety (hence locking) isn't needed.

connect()
Connects to the server.

disconnect()
Disconnects from the server.

is_connected()
Determines whether the sender is connected or not.

Returns true if the sender is connected.

receive(*update_state*)
Receives a single message from the connected endpoint.

This method returns nothing, and will only affect other parts of the MtProtoSender such as the updates callback being fired or a pending request being confirmed.

Any unhandled object (likely updates) will be passed to update_state.process(TLObject).

Parameters **update_state** – the UpdateState that will process all the received Update and Updates objects.

send(**requests*)
Sends the specified TLObject(s) (which must be requests), and acknowledging any message which needed confirmation.

Parameters **requests** – the requests to be sent.

telethon.tl package

telethon.tl package

telethon.tl.custom package

telethon.tl.custom.draft module

class telethon.tl.custom.draft.**Draft**(*client*, *peer*, *draft*)
Bases: object

Custom class that encapsulates a draft on the Telegram servers, providing an abstraction to change the message conveniently. The library will return instances of this class when calling get_drafts().

Args:

date (datetime): The date of the draft.

link_preview (bool): Whether the link preview is enabled or not.

reply_to_msg_id (int): The message ID that the draft will reply to.

delete()
Deletes this draft, and returns True on success.

entity
The entity that belongs to this dialog (user, chat or channel).

input_entity
Input version of the entity.

is_empty
Convenience bool to determine if the draft is empty or not.

raw_text
The raw (text without formatting) contained in the draft. It will be empty if there is no text (thus draft not set).

send(*clear=True*, *parse_mode='md'*)
Sends the contents of this draft to the dialog. This is just a wrapper around send_message(dialog.input_entity, *args, **kwargs).

set_message(*text=None*, *reply_to=0*, *parse_mode='md'*, *link_preview=None*)
Changes the draft message on the Telegram servers. The changes are reflected in this object.

Parameters

- **text** (*str*) – New text of the draft. Preserved if left as None.
- **reply_to** (*int*) – Message ID to reply to. Preserved if left as 0, erased if set to None.
- **link_preview** (*bool*) – Whether to attach a web page preview. Preserved if left as None.
- **parse_mode** (*str*) – The parse mode to be used for the text.

Return bool `True` on success.

stringify()

text
The markdown text contained in the draft. It will be empty if there is no text (and hence no draft is set).

to_dict()

telethon.tl.custom.dialog module

class telethon.tl.custom.dialog.**Dialog**(*client*, *dialog*, *entities*, *messages*)
Bases: object

Custom class that encapsulates a dialog (an open "conversation" with someone, a group or a channel) providing an abstraction to easily access the input version/normal entity/message etc. The library will return instances of this class when calling *get_dialogs()*.

Args:

dialog (Dialog): The original `Dialog` instance.

pinned (bool): Whether this dialog is pinned to the top or not.

message (Message): The last message sent on this dialog. Note that this member will not be updated when new messages arrive, it's only set on creation of the instance.

date (datetime): The date of the last message sent on this dialog.

entity (entity): The entity that belongs to this dialog (user, chat or channel).

input_entity (InputPeer): Input version of the entity.

id (int): The marked ID of the entity, which is guaranteed to be unique.

name (str): Display name for this dialog. For chats and channels this is their title, and for users it's "First-Name Last-Name".

unread_count (int): How many messages are currently unread in this dialog. Note that this value won't update when new messages arrive.

unread_mentions_count (int): How many mentions are currently unread in this dialog. Note that this value won't update when new messages arrive.

draft (*telethon.tl.custom.draft.Draft*): The draft object in this dialog. It will not be `None`, so you can call `draft.set_message(...)`.

send_message(**args*, ***kwargs*)
Sends a message to this dialog. This is just a wrapper around `client.send_message(dialog.input_entity, *args, **kwargs)`.

stringify()

to_dict()

telethon.tl.gzip_packed module

class telethon.tl.gzip_packed.**GzipPacked**(*data*)
Bases: *telethon.tl.tlobject.TLObject*

CONSTRUCTOR_ID = 812830625

static gzip_if_smaller(*request*)
Calls bytes(request), and based on a certain threshold, optionally gzips the resulting data. If the gzipped data is smaller than the original byte array, this is returned instead.

Note that this only applies to content related requests.

static read(*reader*)

telethon.tl.message_container module

class telethon.tl.message_container.**MessageContainer**(*messages*)
Bases: *telethon.tl.tlobject.TLObject*

CONSTRUCTOR_ID = 1945237724

static iter_read(*reader*)

stringify()

to_dict(*recursive=True*)

telethon.tl.tl_message module

class telethon.tl.tl_message.**TLMessage**(*session*, *request*)
Bases: *telethon.tl.tlobject.TLObject*

https://core.telegram.org/mtproto/service_messages#simple-container

stringify()

to_dict(*recursive=True*)

telethon.tl.tlobject module

class telethon.tl.tlobject.**TLObject**
Bases: object

static from_reader(*reader*)

on_response(*reader*)

static pretty_format(*obj*, *indent=None*)
Pretty formats the given object as a string which is returned. If indent is None, a single line will be returned.

resolve(*client*, *utils*)

static serialize_bytes(*data*)
Write bytes by using Telegram guidelines

static serialize_datetime(*dt*)

stringify()

`to_dict`()

Module contents

1.28 Indices and tables

- genindex
- modindex
- search

Python Module Index

t

Index

D

E

F

G

I

K

L

M

N

O

P

R

S

T

U

V

W