Telethon Documentation

Release 1.6.2

Lonami

Mar 23, 2019

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:

- Are you new here? Jump straight into *Getting Started*!
- Looking for available friendly methods? See *telethon.client package*.
- Used Telethon before v1.0? See *Compatibility and Convenience*.
- Need the full API reference? https://lonamiwebs.github.io/Telethon/.

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

Contents

1.1.1 Simple Installation

```
pip3 install telethon
```

More details: *Installation*

1.1.2 Creating a client

```
from telethon import TelegramClient, sync

# 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).start()
```

More details: *Creating a Client*

1.1.3 Basic Usage

```
# Getting information about yourself
me = client.get_me()
print(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.iter_messages('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(dialog.name, dialog.draft.text)

# 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(),
# or even using message.download_media():
messages = client.get_messages('username')
messages[0].download_media()
```

More details: *TelegramClient*

See *telethon.client package* for all available friendly methods.

1.1.4 Handling Updates

```
from telethon import events

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

client.run_until_disconnected()
```

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

Contents

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:

```
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
```

5. Done!

To generate the method documentation, `python3 setup.py gen docs`.

1.2.3 Optional dependencies

If pillow is installed, large images will be automatically resized when sending photos to prevent Telegram from failing with "invalid image". Official clients also do this.

If aiohttp is installed, the library will be able to download WebDocument media files (otherwise you will get an error).

If hachoir is installed, it will be used to extract metadata from files when sending documents. Telegram uses this information to show the song's performer, artist, title, duration, and for videos too (including size). Otherwise, they will default to empty values, and you can set the attributes manually.

If cryptg is installed, encryption and decryption will be made in C instead of Python which will be a lot faster. If your code deals with a lot of updates or you are downloading/uploading a lot of files, you will notice a considerable speed-up (from a hundred kilobytes per second to several megabytes per second, if your connection allows it). If it's not installed, pyaes will be used (which is pure Python, so it's much slower).

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, sync

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

(continues on next page)

(continued from previous page)

```
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`.

Note: It's important that the library always accesses the same session file so that you don't need to re-send the code over and over again. By default it creates the file in your working directory, but absolute paths work too.

Once you have a client ready, simply *`.start()`* it:

```
client.start()
```

This line connects to Telegram, checks whether the current user is authorized or not, and if it's not, it begins the login or sign up process.

When you're done with your code, you should always disconnect:

```
client = TelegramClient(...)
try:
    client.start()
    ...  # your code here
finally:
    client.disconnect()
```

You can also use a `with` block to achieve the same effect:

```
client = TelegramClient(...)
with client:
    ...  # your code here

# or
with TelegramClient(...) as client:
    ...  # your code here
```

Wrapping it all together:

```
from telethon import TelegramClient, sync
with TelegramClient('session_name', api_id, api_hash) as client:
    ...  # your code
```

Just two setup lines.

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',
```

(continues on next page)

(continued from previous page)

```
    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 Manually Signing In

Note: Skip this unless you need more control when connecting.

If you need more control, you can replicate what *`client.start()`* is doing behind the scenes for your convenience. The first step is to connect to the servers:

```
client.connect()
```

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:

```
from telethon import TelegramClient, sync
client = TelegramClient('session_name', api_id, api_hash)

client.connect()
if not client.is_user_authorized():
    client.send_code_request(phone_number)
    me = client.sign_in(phone_number, input('Enter code: '))
```

Remember that this is the manual process and it's so much easier to use the code snippets shown at the beginning of the page.

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()`*.

Two Factor Authorization (2FA)

If you have Two Factor Authorization (from now on, 2FA) enabled on your account, calling *`.sign_in()`* will raise a `SessionPasswordNeededError`. When this happens, just use the method 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

Note: Make sure to use the friendly methods described in *telethon.client package*! This section is just an introduction to using the client, but all the available methods are in the *telethon.client package* reference, including detailed descriptions to what they do.

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 (assuming you have `from telethon import sync` or `import telethon.sync`):

```
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 a 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 a 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.iter_messages('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):
```

(continues on next page)

(continued from previous page)

```
    print(dialog.name, dialog.draft.text)

# 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.

1.4.1 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.client package* to find the available methods.

1.5 Users, Chats and Channels

Important: TL;DR; If you're here because of *"Could not find the input entity for"*, you must ask yourself "how did I find this entity through official applications"? Now do the same with the library. Use what applies:

```
with client:
    # Does it have an username? Use it!
    entity = client.get_entity(username)

    # Do you have a conversation open with them? Get dialogs.
    client.get_dialogs()

    # Are they participant of some group? Get them.
    client.get_participants('TelethonChat')

    # Is the entity the original sender of a forwarded message? Get it.
    client.get_messages('TelethonChat', 100)

    # NOW you can use the ID, anywhere!
    entity = client.get_entity(123456)
    client.send_message(123456, 'Hi!')
```

Once the library has "seen" the entity, you can use their **integer** ID. You can't use entities from IDs the library hasn't seen. You must make the library see them *at least once* and disconnect properly. You know where the entities are and you must tell the library. It won't guess for you.

Contents

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 contact list**.

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)`*.

Once you have encountered an ID, the library will (by default) have saved their `access_hash` for you, which is needed to invoke most methods. This is why sometimes you might encounter this error when working with the library. You should `except ValueError` and run code that you know should work to find the entity.

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))
```

(continues on next page)

(continued from previous page)

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

Note: You **don't** need to get the entity before using it! Just let the library do its job. Use a phone from your contacts, username, ID or input entity (preferred but not necessary), whatever you already have.

All methods in the *TelegramClient* call *`.get_input_entity()`* prior to sending the request 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 *`client.get_entity()`* beforehand, just use the username, a phone from your contacts, or the entity retrieved by other means like *`client.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**. They are named like this because they are input parameters in the requests.

Entities' ID are the same for all user and bot accounts, however, the access hash is **different for each account**, so trying to reuse the access hash from one account in another will **not** work.

Sometimes, Telegram only needs to indicate the type of the entity along with their ID. For this purpose, Peer versions of the entities also exist, which just have the ID. You cannot get the hash out of them since you should not be needing it. The library probably has cached it before.

Peers are enough to identify an entity, but they are not enough to make a request with them use them. You need to know their hash before you can "use them", and to know the hash you need to "encounter" them, let it be in your dialogs, participants, message forwards, etc.

Note: You *can* use peers with the library. Behind the scenes, they are replaced with the input variant. Peers "aren't enough" on their own but the library will do some more work to use the right type.

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, *`client.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 *`client.get_input_entity()`* **over** *`client.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 *`client.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 *`client.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.

1.5.4 Full entities

In addition to PeerUser, InputPeerUser, User (and its variants for chats and channels), there is also the concept of UserFull.

This full variant has additional information such as whether the user is blocked, its notification settings, the bio or about of the user, etc.

There is also messages.ChatFull which is the equivalent of full entities for chats and channels, with also the about section of the channel. Note that the `users` field only contains bots for the channel (so that clients can suggest commands to use).

You can get both of these by invoking GetFullUser, GetFullChat and GetFullChannel respectively.

1.6 Working with Updates

Important: Coming from Telethon before it reached its version 1.0? Make sure to read *Compatibility and Convenience*! Otherwise, you can ignore this note and just follow along.

The library comes with the `telethon.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!

Important: 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 when working with events, at least the error level, to see if this is happening so you can debug the error.

When using updates, please enable logging:

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

Contents

1.6.1 Getting Started

```
from telethon import TelegramClient, events

client = TelegramClient('name', api_id, api_hash)

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

client.start()
client.run_until_disconnected()
```

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

```
from telethon import TelegramClient, events

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

This is normal creation (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:

```
async def my_event_handler(event):
    if 'hello' in event.raw_text:
        await 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.

Do you notice anything different? Yes! Event handlers **must** be `async` for them to work, and **every method using the network** needs to have an `await`, otherwise, Python's `asyncio` will tell you that you forgot to do so, so you can easily add it.

```
client.start()
client.run_until_disconnected()
```

Finally, this tells the client that we're done with our code. We run the `asyncio` loop until the client starts (this is done behind the scenes, since the method is so common), and then we run it again until we are disconnected. Of course, you can do other things instead of running until disconnected. 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 *event.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')))
async def normal_handler(event):
    if 'roll' in event.raw_text:
        await 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,
                             pattern='eval (.+)'))
async def admin_handler(event):
    expression = event.pattern_match.group(1)
    await 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 Properties vs. Methods

The event shown above acts just like a *custom.Message*, which means you can access all the properties it has, like `.sender`.

However events are different to other methods in the client, like *client.get_messages*. Events *may not* send information about the sender or chat, which means it can be `None`, but all the methods defined in the client always have this information so it doesn't need to be re-fetched. For this reason, you have `get_` methods, which will make a network call if necessary.

In short, you should do this:

```
@client.on(events.NewMessage)
async def handler(event):
    # event.input_chat may be None, use event.get_input_chat()
    chat = await event.get_input_chat()
    sender = await event.get_sender()
    buttons = await event.get_buttons()

async def main():
    async for message in client.iter_messages('me', 10):
        # Methods from the client always have these properties ready
        chat = message.input_chat
        sender = message.sender
        buttons = message.buttons
```

Notice, properties (`message.sender`) don't need an `await`, but methods (`message.get_sender`) **do** need an `await`, and you should use methods in events for these properties that may need network.

1.6.4 Events Without the client

The code of your application starts getting big, so you decide to separate the handlers into different files. But how can you access the client from these files? You don't need to! Just *events.register* them:

```
# handlers/welcome.py
from telethon import events

@events.register(events.NewMessage('(?i)hello'))
async def handler(event):
    client = event.client
    await event.respond('Hey!')
    await client.send_message('me', 'I said hello to someone')
```

Registering events is a way of saying "this method is an event handler". You can use *telethon.events.is_handler* to check if any method is a handler. You can think of them as a different approach to Flask's blueprints.

It's important to note that this does **not** add the handler to any client! You never specified the client on which the handler should be used. You only declared that it is a handler, and its type.

To actually use the handler, you need to *client.add_event_handler* to the client (or clients) where they should be added to:

```
# main.py
from telethon import TelegramClient
import handlers.welcome

with TelegramClient(...) as client:
    client.add_event_handler(handlers.welcome.handler)
    client.run_until_disconnected()
```

This also means that you can register an event handler once and then add it to many clients without re-declaring the event.

1.6.5 Events Without Decorators

If for any reason you don't want to use *telethon.events.register*, you can explicitly pass the event handler to use to the mentioned *client.add_event_handler*:

```
from telethon import TelegramClient, events

async def handler(event):
    ...

with TelegramClient(...) as client:
    client.add_event_handler(handler, events.NewMessage)
    client.run_until_disconnected()
```

Similarly, you also have *client.remove_event_handler* and *client.list_event_handlers*.

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

Note: The `event` type is ignored in *client.add_event_handler* if you have used *telethon.events.register* on the `callback` before, since that's the point of using such method at all.

1.6.6 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 *telethon.events.StopPropagation* exception which will cause the propagation of the update through your handlers to stop:

```
from telethon.events import StopPropagation

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

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

@client.on(events.NewMessage)
async 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 Compatibility and Convenience

Telethon is an `asyncio` library. Compatibility is an important concern, and while it can't always be kept and mistakes happens, the *Changelog (Version History)* is there to tell you when these important changes happen.

Contents

1.7.1 Compatibility

Some decisions when developing will inevitable be proven wrong in the future. One of these decisions was using threads. Now that Python 3.4 is reaching EOL and using `asyncio` is usable as of Python 3.5 it makes sense for a library like Telethon to make a good use of it.

If you have old code, **just use old versions** of the library! There is nothing wrong with that other than not getting new updates or fixes, but using a fixed version with `pip install telethon==0.19.1.6` is easy enough to do.

You might want to consider using Virtual Environments in your projects.

There's no point in maintaining a synchronous version because the whole point is that people don't have time to upgrade, and there has been several changes and clean-ups. Using an older version is the right way to go.

Sometimes, other small decisions are made. These all will be reflected in the *Changelog (Version History)* which you should read when upgrading.

If you want to jump the `asyncio` boat, here are some of the things you will need to start migrating really old code:

```
# 1. Import the client from telethon.sync
from telethon.sync import TelegramClient

# 2. Change this monster...
try:
    assert client.connect()
    if not client.is_user_authorized():
        client.send_code_request(phone_number)
        me = client.sign_in(phone_number, input('Enter code: '))

    ...  # REST OF YOUR CODE
finally:
    client.disconnect()

# ...for this:
with client:
    ...  # REST OF YOUR CODE

# 3. client.idle() no longer exists.
# Change this...
client.idle()
# ...to this:
client.run_until_disconnected()

# 4. client.add_update_handler no longer exists.
# Change this...
client.add_update_handler(handler)
# ...to this:
client.add_event_handler(handler)
```

In addition, all the update handlers must be `async def`, and you need to `await` method calls that rely on network requests, such as getting the chat or sender. If you don't use updates, you're done!

1.7.2 Convenience

Note: The entire documentation assumes you have done one of the following:

```
from telethon import TelegramClient, sync
# or
from telethon.sync import TelegramClient
```

This makes the examples shorter and easier to think about.

For quick scripts that don't need updates, it's a lot more convenient to forget about `asyncio` and just work with sequential code. This can prove to be a powerful hybrid for running under the Python REPL too.

```
from telethon.sync import TelegramClient
#             ^~~~~ note this part; it will manage the asyncio loop for you

with TelegramClient(...) as client:
```

(continues on next page)

(continued from previous page)

```
    print(client.get_me().username)
    #     ^ notice the lack of await, or loop.run_until_complete().
    #       Since there is no loop running, this is done behind the scenes.
    #
    message = client.send_message('me', 'Hi!')
    import time
    time.sleep(5)
    message.delete()

    # You can also have an hybrid between a synchronous
    # part and asynchronous event handlers.
    #
    from telethon import events
    @client.on(events.NewMessage(pattern='(?i)hi|hello'))
    async def handler(event):
        await event.reply('hey')

    client.run_until_disconnected()
```

Some methods, such as `with`, `start`, `disconnect` and `run_until_disconnected` work both in synchronous and asynchronous contexts by default for convenience, and to avoid the little overhead it has when using methods like sending a message, getting messages, etc. This keeps the best of both worlds as a sane default.

Note: As a rule of thumb, if you're inside an `async def` and you need the client, you need to `await` calls to the API. If you call other functions that also need API calls, make them `async def` and `await` them too. Otherwise, there is no need to do so with this mode.

1.7.3 Speed

When you're ready to micro-optimize your application, or if you simply don't need to call any non-basic methods from a synchronous context, just get rid of `telethon.sync` and work inside an `async def`:

```
import asyncio
from telethon import TelegramClient, events

async def main():
    async with TelegramClient(...) as client:
        print((await client.get_me()).username)
        #     ^_____________________^ notice these parenthesis
        #     You want to ``await`` the call, not the username.
        #
        message = await client.send_message('me', 'Hi!')
        await asyncio.sleep(5)
        await message.delete()

        @client.on(events.NewMessage(pattern='(?i)hi|hello'))
        async def handler(event):
            await event.reply('hey')

        await client.run_until_disconnected()

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
```

The `telethon.sync` magic module simply wraps every method behind:

```
loop = asyncio.get_event_loop()
loop.run_until_complete(main())
```

So that you don't have to write it yourself every time. That's the overhead you pay if you import it, and what you save if you don't.

1.7.4 Learning

You know the library uses `asyncio` everywhere, and you want to learn how to do things right. Even though `asyncio` is its own topic, the documentation wants you to learn how to use Telethon correctly, and for that, you need to use `asyncio` correctly too. For this reason, there is a section called *Mastering asyncio* that will introduce you to the `asyncio` world, with links to more resources for learning how to use it. Feel free to check that section out once you have read the rest.

1.8 Accessing the Full API

Important: While you have access to this, you should always use the friendly methods listed on *telethon.client package* unless you have a better reason not to, like a method not existing or you wanting more control.

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.

Important: All the examples in this documentation assume that you have `from telethon import sync` or `import telethon.sync` for the sake of simplicity and that you understand what it does (see *Compatibility and Convenience* for more). Simply add either line at the beginning of your project and it will 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 *client.get_input_entity*:

```
import telethon.sync
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 *client.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 *client.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 *telethon.utils.get_input_peer*:

```
from telethon import utils
peer = utils.get_input_peer(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 *client.get_entity* versus *client.get_input_entity*, we have everything we need. To invoke our request we do:

```
result = client(SendMessageRequest(peer, 'Hello there!'))
```

Message sent! Of course, this is only an example. There are over 250 methods available as of layer 80, 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.

1.8.1 Requests in Parallel

The library will automatically merge outgoing requests into a single *container*. Telegram's API supports sending multiple requests in a single container, which is faster because it has less overhead and the server can run them without waiting for others. You can also force using a container manually:

```
async def main():

    # Letting the library do it behind the scenes
    await asyncio.wait([
        client.send_message('me', 'Hello'),
        client.send_message('me', ','),
        client.send_message('me', 'World'),
        client.send_message('me', '.')
    ])

    # Manually invoking many requests at once
    await client([
        SendMessageRequest('me', 'Hello'),
        SendMessageRequest('me', ', '),
        SendMessageRequest('me', 'World'),
        SendMessageRequest('me', '.')
    ])
```

Note that you cannot guarantee the order in which they are run. Try running the above code more than one time. You will see the order in which the messages arrive is different.

If you use the raw API (the first option), you can use `ordered` to tell the server that it should run the requests sequentially. This will still be faster than going one by one, since the server knows all requests directly:

```
client([
    SendMessageRequest('me', 'Hello'),
    SendMessageRequest('me', ', '),
    SendMessageRequest('me', 'World'),
    SendMessageRequest('me', '.')
], ordered=True)
```

If any of the requests fails with a Telegram error (not connection errors or any other unexpected events), the library will raise *telethon.errors.common.MultiError*. You can `except` this and still access the successful results:

```
from telethon.errors import MultiError

try:
    client([
        SendMessageRequest('me', 'Hello'),
        SendMessageRequest('me', ''),
        SendMessageRequest('me', 'World')
    ], ordered=True)
except MultiError as e:
    # The first and third requests worked.
    first = e.results[0]
    third = e.results[2]
```

(continues on next page)

(continued from previous page)

```
    # The second request failed.
    second = e.exceptions[1]
```

1.9 Session Files

Contents

1.9.1 What are sessions?

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 in the working directory.

Note that if you pass a string it will be a file in the current working directory, although you can also pass absolute paths.

The session file contains enough information for you to login without re-sending the code, so if you have to enter the code more than once, maybe you're changing the working directory, renaming or removing the file, or using random names.

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 a 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`.

1.9.2 Different 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 three implementations of the abstract `Session` class:

- `MemorySession`: stores session data within memory.

- `SQLiteSession`: stores sessions within on-disk SQLite databases. Default.
- `StringSession`: stores session data within memory, but can be saved as a string.

You can import these `from telethon.sessions`. For example, using the `StringSession` is done as follows:

```
from telethon.sync import TelegramClient
from telethon.sessions import StringSession

with TelegramClient(StringSession(string), api_id, api_hash) as client:
    ...  # use the client

    # Save the string session as a string; you should decide how
    # you want to save this information (over a socket, remote
    # database, print it and then paste the string in the code,
    # etc.); the advantage is that you don't need to save it
    # on the current disk as a separate file, and can be reused
    # anywhere else once you log in.
    string = client.session.save()

# Note that it's also possible to save any other session type
# as a string by using ``StringSession.save(session_instance)``:
client = TelegramClient('sqlite-session', api_id, api_hash)
string = StringSession.save(client.session)
```

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.

1.9.3 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.

1.9.4 String Sessions

`StringSession` are a convenient way to embed your login credentials directly into your code for extremely easy portability, since all they take is a string to be able to login without asking for your phone and code (or faster start if you're using a bot token).

The easiest way to generate a string session is as follows:

```
from telethon.sync import TelegramClient
from telethon.sessions import StringSession

with TelegramClient(StringSession(), api_id, api_hash) as client:
    print(client.session.save())
```

Think of this as a way to export your authorization key (what's needed to login into your account). This will print a string in the standard output (likely your terminal).

> **Warning:** **Keep this string safe!** Anyone with this string can use it to login into your account and do anything they want to to do.
>
> This is similar to leaking your `*.session` files online, but it is easier to leak a string than it is to leak a file.

Once you have the string (which is a bit long), load it into your script somehow. You can use a normal text file and `open(...).read()` it or you can save it in a variable directly:

```
string = '1aaNk8EX-YRfwoRsebUkugFvht6DUPi_Q25UOCzOAqzc...'
with TelegramClient(StringSession(string), api_id, api_hash) as client:
    client.send_message('me', 'Hi')
```

These strings are really convenient for using in places like Heroku since their ephemeral filesystem will delete external files once your application is over.

1.10 Update Modes

With `asyncio`, the library has several tasks running in the background. One task is used for sending requests, another task is used to receive them, and a third one is used to handle updates.

To handle updates, you must keep your script running. You can do this in several ways. For instance, if you are *not* running `asyncio`'s event loop, you should use *`client.run_until_disconnected`*:

```
import asyncio
from telethon import TelegramClient

client = TelegramClient(...)
...
client.run_until_disconnected()
```

Behind the scenes, this method is `await`'ing on the *`client.disconnected`* property, so the code above and the following are equivalent:

```
import asyncio
from telethon import TelegramClient

client = TelegramClient(...)

async def main():
    await client.disconnected

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
```

You could also run *`client.disconnected`* until it completed.

But if you don't want to `await`, then you should know what you want to be doing instead! What matters is that you shouldn't let your script die. If you don't care about updates, you don't need any of this.

Notice that unlike *`client.disconnected`*, *`client.run_until_disconnected`* will handle `KeyboardInterrupt` with you. This method is special and can also be ran while the loop is running, so you can do this:

```
async def main():
    await client.run_until_disconnected()
```

(continues on next page)

(continued from previous page)

```
loop.run_until_complete(main())
```

If you need to process updates sequentially (i.e. not in parallel), you should set `sequential_updates=True` when creating the client:

```
with TelegramClient(..., sequential_updates=True) as client:
    ...
```

1.11 Mastering Telethon

You've come far! In this section you will learn best practices, as well as how to fix some silly (yet common) errors you may have found. Let's start with a simple one.

1.11.1 Asyncio madness

We promise `asyncio` is worth learning. Take your time to learn it. It's a powerful tool that enables you to use this powerful library. You need to be comfortable with it if you want to master Telethon.

```
AttributeError: 'coroutine' object has no attribute 'id'
```

You probably had a previous version, upgraded, and expected everything to work. Remember, just add this line:

```
import telethon.sync
```

If you're inside an event handler you need to `await` **everything** that *makes a network request*. Getting users, sending messages, and nearly everything in the library needs access to the network, so they need to be awaited:

```
@client.on(events.NewMessage)
async def handler(event):
    print((await event.get_sender()).username)
```

You may want to read https://lonamiwebs.github.io/blog/asyncio/ to help you understand `asyncio` better. I'm open for feedback regarding that blog post

1.11.2 Entities

A lot of methods and requests require *entities* to work. For example, you send a message to an *entity*, get the username of an *entity*, and so on. There is an entire section on this at *Users, Chats and Channels* due to their importance.

There are a lot of things that work as entities: usernames, phone numbers, chat links, invite links, IDs, and the types themselves. That is, you can use any of those when you see an "entity" is needed.

Note: Remember that the phone number must be in your contact list before you can use it.

You should use, **from better to worse**:

1. Input entities. For example, *`event.input_chat`*, *`message.input_sender`*, or caching an entity you will use a lot with `entity = await client.get_input_entity(...)`.

2. Entities. For example, if you had to get someone's username, you can just use `user` or `channel`. It will work. Only use this option if you already have the entity!
3. IDs. This will always look the entity up from the cache (the `*.session` file caches seen entities).
4. Usernames, phone numbers and links. The cache will be used too (unless you force a *client.get_entity()*), but may make a request if the username, phone or link has not been found yet.

In short, unlike in most bot API libraries where you use the ID, you **should not** use the ID *if* you have the input entity. This is OK:

```
async def handler(event):
    await client.send_message(event.sender_id, 'Hi')
```

However, **this is better**:

```
async def handler(event):
    await client.send_message(event.input_sender, 'Hi')
```

Note that this also works for *message* instead of `event`. Telegram may not send the sender information, so if you want to be 99% confident that the above will work you should do this:

```
async def handler(event):
    await client.send_message(await event.get_input_sender(), 'Hi')
```

Methods are able to make network requests to get information that could be missing. Properties will never make a network request.

Of course, it is convenient to IDs or usernames for most purposes. It will be fast enough and caching with *client.get_input_entity(...)* will be a micro-optimization. However it's worth knowing, and it will also let you know if the entity cannot be found beforehand.

Note: Sometimes Telegram doesn't send the access hash inside entities, so using *chat* or *sender* may not work, but *input_chat* and *input_sender* while making requests definitely will since that's what they exist for. If Telegram did not send information about the access hash, you will get something like "Invalid channel object" or "Invalid user object".

1.11.3 Debugging

Please enable logging:

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

Change it for `logging.DEBUG` if you are asked for logs. It will save you a lot of headaches and time when you work with events. This is for errors.

Debugging is *really* important. Telegram's API is really big and there is a lot of things that you should know. Such as, what attributes or fields does a result have? Well, the easiest thing to do is printing it:

```
user = client.get_entity('Lonami')
print(user)
```

That will show a huge line similar to the following:

```
User(id=10885151, is_self=False, contact=False, mutual_contact=False, deleted=False,␣
↪bot=False, bot_chat_history=False, bot_nochats=False, verified=False,␣
↪restricted=False, min=False, bot_inline_geo=False, access_hash=123456789012345678,␣
↪first_name='Lonami', last_name=None, username='Lonami', phone=None,␣
↪photo=UserProfilePhoto(photo_id=123456789012345678, photo_small=FileLocation(dc_
↪id=4, volume_id=1234567890, local_id=1234567890, secret=123456789012345678), photo_
↪big=FileLocation(dc_id=4, volume_id=1234567890, local_id=1234567890,␣
↪secret=123456789012345678)), status=UserStatusOffline(was_online=datetime.
↪datetime(2018, 1, 2, 3, 4, 5, tzinfo=datetime.timezone.utc)), bot_info_version=None,
↪ restriction_reason=None, bot_inline_placeholder=None, lang_code=None)
```

That's a lot of text. But as you can see, all the properties are there. So if you want the username you **don't use regex** or anything like splitting `str(user)` to get what you want. You just access the attribute you need:

```
username = user.username
```

Can we get better than the shown string, though? Yes!

```
print(user.stringify())
```

Will show a much better:

```
User(
    id=10885151,
    is_self=False,
    contact=False,
    mutual_contact=False,
    deleted=False,
    bot=False,
    bot_chat_history=False,
    bot_nochats=False,
    verified=False,
    restricted=False,
    min=False,
    bot_inline_geo=False,
    access_hash=123456789012345678,
    first_name='Lonami',
    last_name=None,
    username='Lonami',
    phone=None,
    photo=UserProfilePhoto(
        photo_id=123456789012345678,
        photo_small=FileLocation(
            dc_id=4,
            volume_id=123456789,
            local_id=123456789,
            secret=-123456789012345678
        ),
        photo_big=FileLocation(
            dc_id=4,
            volume_id=123456789,
            local_id=123456789,
            secret=123456789012345678
        )
    ),
    status=UserStatusOffline(
        was_online=datetime.datetime(2018, 1, 2, 3, 4, 5, tzinfo=datetime.timezone.
↪utc)
```

(continues on next page)

(continued from previous page)

```
    ),
    bot_info_version=None,
    restriction_reason=None,
    bot_inline_placeholder=None,
    lang_code=None
)
```

Now it's easy to see how we could get, for example, the `was_online` time. It's inside `status`:

```
online_at = user.status.was_online
```

You don't need to print everything to see what all the possible values can be. You can just search in http://lonamiwebs.github.io/Telethon/.

Remember that you can use Python's isinstance to check the type of something. For example:

```
from telethon import types

if isinstance(user.status, types.UserStatusOffline):
    print(user.status.was_online)
```

1.11.4 Avoiding Limits

Don't spam. You won't get `FloodWaitError` or your account banned or deleted if you use the library *for legit use cases*. Make cool tools. Don't spam! Nobody knows the exact limits for all requests since they depend on a lot of factors, so don't bother asking.

Still, if you do have a legit use case and still get those errors, the library will automatically sleep when they are smaller than 60 seconds by default. You can set different "auto-sleep" thresholds:

```
client.flood_sleep_threshold = 0  # Don't auto-sleep
client.flood_sleep_threshold = 24 * 60 * 60  # Sleep always
```

You can also except it and act as you prefer:

```
from telethon.errors import FloodWaitError
try:
    ...
except FloodWaitError as e:
    print('Flood waited for', e.seconds)
    quit(1)
```

VoIP numbers are very limited, and some countries are more limited too.

1.11.5 Chat or User From Messages

Although it's explicitly noted in the documentation that messages *subclass* *`ChatGetter`* and *`SenderGetter`*, some people still don't get inheritance.

When the documentation says "Bases: *`telethon.tl.custom.chatgetter.ChatGetter`*" it means that the class you're looking at, *also* can act as the class it bases. In this case, *`ChatGetter`* knows how to get the *chat* where a thing belongs to.

So, a *`Message`* is a *`ChatGetter`*. That means you can do this:

```
message.is_private
message.chat_id
message.get_chat()
# ...etc
```

SenderGetter is similar:

```
message.user_id
message.get_input_user()
message.user
# ...etc
```

Quite a few things implement them, so it makes sense to reuse the code. For example, all events (except raw updates) implement *ChatGetter* since all events occur in some chat.

1.11.6 Session Files

They are an important part for the library to be efficient, such as caching and handling your authorization key (or you would have to login every time!).

However, some people have a lot of trouble with SQLite, especially in Windows:

```
...some lines of traceback
'insert or replace into entities values (?,?,?,?,?)', rows)
sqlite3.OperationalError: database is locked
```

This error occurs when **two or more clients use the same session**, that is, when you write the same session name to be used in the client:

- You have two scripts running (interactive sessions count too).
- You have two clients in the same script running at the same time.

The solution is, if you need two clients, use two sessions. If the problem persists and you're on Linux, you can use `fuser my.session` to find out the process locking the file. As a last resort, you can reboot your system.

If you really dislike SQLite, use a different session storage. There is an entire section covering that at *Session Files*.

1.11.7 Final Words

Now you are aware of some common errors and use cases, this should help you master your Telethon skills to get the most out of the library. Have fun developing awesome things!

1.12 Mastering asyncio

Contents

1.12.1 What's asyncio?

asyncio is a Python 3's built-in library. This means it's already installed if you have Python 3. Since Python 3.5, it is convenient to work with asynchronous code. Before (Python 3.4) we didn't have `async` or `await`, but now we do.

asyncio stands for *Asynchronous Input Output*. This is a very powerful concept to use whenever you work IO. Interacting with the web or external APIs such as Telegram's makes a lot of sense this way.

1.12.2 Why asyncio?

Asynchronous IO makes a lot of sense in a library like Telethon. You send a request to the server (such as "get some message"), and thanks to asyncio, your code won't block while a response arrives.

The alternative would be to spawn a thread for each update so that other code can run while the response arrives. That is *a lot* more expensive.

The code will also run faster, because instead of switching back and forth between the OS and your script, your script can handle it all. Avoiding switching saves quite a bit of time, in Python or any other language that supports asynchronous IO. It will also be cheaper, because tasks are smaller than threads, which are smaller than processes.

1.12.3 What are asyncio basics?

```
# First we need the asyncio library
import asyncio

# Then we need a loop to work with
loop = asyncio.get_event_loop()

# We also need something to run
async def main():
    for char in 'Hello, world!\n':
        print(char, end='', flush=True)
        await asyncio.sleep(0.2)

# Then, we need to run the loop with a task
loop.run_until_complete(main())
```

1.12.4 What does telethon.sync do?

The moment you import any of these:

```
from telethon import sync, ...
# or
from telethon.sync import ...
# or
import telethon.sync
```

The `sync` module rewrites most `async def` methods in Telethon to something similar to this:

```
def new_method():
    result = original_method()
    if loop.is_running():
        # the loop is already running, return the await-able to the user
        return result
    else:
        # the loop is not running yet, so we can run it for the user
        return loop.run_until_complete(result)
```

That means you can do this:

```
print(client.get_me().username)
```

Instead of this:

```
import asyncio
loop = asyncio.get_event_loop()
me = loop.run_until_complete(client.get_me())
print(me.username)
```

As you can see, it's a lot of boilerplate and noise having to type `run_until_complete` all the time, so you can let the magic module to rewrite it for you. But notice the comment above: it won't run the loop if it's already running, because it can't. That means this:

```
async def main():
    # 3. the loop is running here
    print(
        client.get_me()  # 4. this will return a coroutine!
        .username  # 5. this fails, coroutines don't have usernames
    )

loop.run_until_complete(  # 2. run the loop and the ``main()`` coroutine
    main()  # 1. calling ``async def`` "returns" a coroutine
)
```

Will fail. So if you're inside an `async def`, then the loop is running, and if the loop is running, you must `await` things yourself:

```
async def main():
    print((await client.get_me()).username)

loop.run_until_complete(main())
```

1.12.5 What are async, await and coroutines?

The `async` keyword lets you define asynchronous functions, also known as coroutines, and also iterate over asynchronous loops or use `async with`:

```
import asyncio

async def main():
    # ^ this declares the main() coroutine function

    async with client:
        # ^ this is an asynchronous with block

        async for message in client.iter_messages(chat):
            # ^ this is a for loop over an asynchronous generator

            print(message.sender.username)

loop = asyncio.get_event_loop()
# ^ this assigns the default event loop from the main thread to a variable

loop.run_until_complete(main())
# ^ this runs the *entire* loop until the main() function finishes.
#   While the main() function does not finish, the loop will be running.
#   While the loop is running, you can't run it again.
```

The `await` keyword blocks the *current* task, and the loop can run other tasks. Tasks can be thought of as "threads", since many can run concurrently:

```
import asyncio

async def hello(delay):
    await asyncio.sleep(delay)  # await tells the loop this task is "busy"
    print('hello')  # eventually the loop resumes the code here

async def world(delay):
    # the loop decides this method should run first
    await asyncio.sleep(delay)  # await tells the loop this task is "busy"
    print('world')  # eventually the loop finishes all tasks

loop = asyncio.get_event_loop()  # get the default loop for the main thread
loop.create_task(world(2))  # create the world task, passing 2 as delay
loop.create_task(hello(delay=1))  # another task, but with delay 1
try:
    # run the event loop forever; ctrl+c to stop it
    # we could also run the loop for three seconds:
    #     loop.run_until_complete(asyncio.sleep(3))
    loop.run_forever()
except KeyboardInterrupt:
    pass
```

The same example, but without the comment noise:

```
import asyncio

async def hello(delay):
    await asyncio.sleep(delay)
    print('hello')

async def world(delay):
    await asyncio.sleep(delay)
    print('world')
```

(continues on next page)

(continued from previous page)

```
loop = asyncio.get_event_loop()
loop.create_task(world(2))
loop.create_task(hello(1))
loop.run_until_complete(asyncio.sleep(3))
```

1.12.6 Can I use threads?

Yes, you can, but you must understand that the loops themselves are not thread safe. and you must be sure to know what is happening. You may want to create a loop in a new thread and make sure to pass it to the client:

```
import asyncio
import threading

def go():
    loop = asyncio.new_event_loop()
    client = TelegramClient(..., loop=loop)
    ...

threading.Thread(target=go).start()
```

Generally, **you don't need threads** unless you know what you're doing. Just create another task, as shown above. If you're using the Telethon with a library that uses threads, you must be careful to use `threading.Lock` whenever you use the client, or enable the compatible mode. For that, see *Compatibility and Convenience*.

You may have seen this error:

```
RuntimeError: There is no current event loop in thread 'Thread-1'.
```

It just means you didn't create a loop for that thread, and if you don't pass a loop when creating the client, it uses `asyncio.get_event_loop()`, which only works in the main thread.

1.12.7 client.run_until_disconnected() blocks!

All of what *`client.run_until_disconnected()`* does is run the asyncio's event loop until the client is disconnected. That means *the loop is running*. And if the loop is running, it will run all the tasks in it. So if you want to run *other* code, create tasks for it:

```
from datetime import datetime

async def clock():
    while True:
        print('The time:', datetime.now())
        await asyncio.sleep(1)

loop.create_task(clock())
...
client.run_until_disconnected()
```

This creates a task for a clock that prints the time every second. You don't need to use *`client.run_until_disconnected()`* either! You just need to make the loop is running, somehow. `asyncio.run_forever` and `asyncio.run_until_complete` can also be used to run the loop, and Telethon will be happy with any approach.

Of course, there are better tools to run code hourly or daily, see below.

1.12.8 What else can asyncio do?

Asynchronous IO is a really powerful tool, as we've seen. There are plenty of other useful libraries that also use asyncio and that you can integrate with Telethon.

- aiohttp is like the infamous requests but asynchronous.
- quart is an asynchronous alternative to Flask.
- aiocron lets you schedule things to run things at a desired time, or run some tasks hourly, daily, etc.

And of course, asyncio itself! It has a lot of methods that let you do nice things. For example, you can run requests in parallel:

```
async def main():
    last, sent, download_path = await asyncio.gather(
        client.get_messages('TelethonChat', 10),
        client.send_message('TelethonOfftopic', 'Hey guys!'),
        client.download_profile_photo('TelethonChat')
    )

loop.run_until_complete(main())
```

This code will get the 10 last messages from @TelethonChat, send one to @TelethonOfftopic, and also download the profile photo of the main group. asyncio will run all these three tasks at the same time. You can run all the tasks you want this way.

A different way would be:

```
loop.create_task(client.get_messages('TelethonChat', 10))
loop.create_task(client.send_message('TelethonOfftopic', 'Hey guys!'))
loop.create_task(client.download_profile_photo('TelethonChat'))
```

They will run in the background as long as the loop is running too.

You can also start an asyncio server in the main script, and from another script, connect to it to achieve Inter-Process Communication. You can get as creative as you want. You can program anything you want. When you use a library, you're not limited to use only its methods. You can combine all the libraries you want. People seem to forget this simple fact!

1.12.9 Why does client.start() work outside async?

Because it's so common that it's really convenient to offer said functionality by default. This means you can set up all your event handlers and start the client without worrying about loops at all.

Using the client in a `with` block, *`start`*, *`run_until_disconnected`*, and *`disconnect`* all support this.

1.12.10 Where can I read more?

Check out my blog post about asyncio, which has some more examples and pictures to help you understand what happens when the loop runs.

1.13 Examples with the Client

This section explores the methods defined in the *TelegramClient* with some practical examples. The section assumes that you have imported the `telethon.sync` package and that you have a client ready to use.

Note: There are some very common errors (such as forgetting to add `import telethon.sync`) for newcomers to `asyncio`:

```
# AttributeError: 'coroutine' object has no attribute 'first_name'
print(client.get_me().first_name)

# TypeError: 'AsyncGenerator' object is not iterable
for message in client.iter_messages('me'):
    ...

# RuntimeError: This event loop is already running
with client.conversation('me') as conv:
    ...
```

That error means you're probably inside an `async def` so you need to use:

```
print((await client.get_me()).first_name)
async for message in client.iter_messages('me'):
    ...

async with client.conversation('me') as conv:
    ...
```

You can of course call other `def` functions from your `async def` event handlers, but if they need making API calls, make your own functions `async def` so you can `await` things:

```
async def helper(client):
    await client.send_message('me', 'Hi')
```

If you're not inside an `async def` you can enter one like so:

```
import asyncio
loop = asyncio.get_event_loop()
loop.run_until_complete(my_async_def())
```

Contents

1.13.1 Authorization

Starting the client is as easy as calling *client.start()*:

```
client.start()
...  # code using the client
client.disconnect()
```

And you can even use a `with` block:

```
with client:
    ... # code using the client
```

Note: Remember we assume you have `import telethon.sync`. You can of course use the library without importing it. The code would be rewritten as:

```
import asyncio
loop = asyncio.get_event_loop()

async def main():
    await client.start()
    ...
    await client.disconnect()

    # or
    async with client:
        ...

loop.run_until_complete(main())
```

All methods that need access to the network (e.g. to make an API call) **must** be awaited (or their equivalent such as `async for` and `async with`). You can do this yourself or you can let the library do it for you by using `import telethon.sync`. With event handlers, you must do this yourself.

The cleanest way to delete your `*.session` file is *client.log_out*. Note that you will obviously need to login again if you use this:

```
# Logs out and deletes the session file; you will need to sign in again
client.log_out()

# You often simply want to disconnect. You will not need to sign in again
client.disconnect()
```

1.13.2 Group Chats

You can easily iterate over all the User in a chat and do anything you want with them by using *client.iter_participants*:

```
for user in client.iter_participants(chat):
    ...  # do something with the user
```

You can also search by their name:

```
for user in client.iter_participants(chat, search='name'):
    ...
```

Or by their type (e.g. if they are admin) with ChannelParticipantsFilter:

```
from telethon.tl.types import ChannelParticipantsAdmins

for user in client.iter_participants(chat, filter=ChannelParticipantsAdmins):
    ...
```

1.13.3 Open Conversations and Joined Channels

The conversations you have open and the channels you have joined are in your "dialogs", so to get them you need to *client.get_dialogs*:

```
dialogs = client.get_dialogs()
first = dialogs[0]
print(first.title)
```

You can then use the dialog as if it were a peer:

```
client.send_message(first, 'hi')
```

You can access *dialog.draft* or you can get them all at once without getting the dialogs:

```
drafts = client.get_drafts()
```

1.13.4 Downloading Media

It's easy to *download_profile_photo*:

```
client.download_profile_photo(user)
```

Or *download_media* from a message:

```
client.download_media(message)
client.download_media(message, filename)
# or
message.download_media()
message.download_media(filename)
```

Remember that these methods return the final filename where the media was downloaded (e.g. it may add the extension automatically).

1.13.5 Getting Messages

You can easily iterate over all the *messages* of a chat with *iter_messages*:

```
for message in client.iter_messages(chat):
    ...  # do something with the message from recent to older

for message in client.iter_messages(chat, reverse=True):
    ...  # going from the oldest to the most recent
```

You can also use it to search for messages from a specific person:

```
for message in client.iter_messages(chat, from_user='me'):
    ...
```

Or you can search by text:

```
for message in client.iter_messages(chat, search='hello'):
    ...
```

Or you can search by media with a MessagesFilter:

```
from telethon.tl.types import InputMessagesFilterPhotos

for message in client.iter_messages(chat, filter=InputMessagesFilterPhotos):
    ...
```

If you want a list instead, use the get variant. The second argument is the limit, and `None` means "get them all":

```
from telethon.tl.types import InputMessagesFilterPhotos

# Get 0 photos and print the total
photos = client.get_messages(chat, 0, filter=InputMessagesFilterPhotos)
print(photos.total)

# Get all the photos
photos = client.get_messages(chat, None, filter=InputMessagesFilterPhotos)
```

Or just some IDs:

```
message_1337 = client.get_messages(chats, ids=1337)
```

1.13.6 Exporting Messages

If you plan on exporting data from your Telegram account, such as the entire message history from your private conversations, chats or channels, or if you plan to download a lot of media, you may prefer to do this within a *takeout* session. Takeout sessions let you export data from your account with lower flood wait limits.

To start a takeout session, simply call *client.takeout()*:

```
from telethon import errors

try:
    with client.takeout() as takeout:
        for message in takeout.iter_messages(chat, wait_time=0):
            ...  # Do something with the message

except errors.TakeoutInitDelayError as e:
    print('Must wait', e.seconds, 'before takeout')
```

Depending on the condition of the session (for example, when it's very young and the method has not been called before), you may or not need to `except errors.TakeoutInitDelayError`. However, it is good practice.

1.13.7 Sending Messages

Just use *send_message*:

```
client.send_message('lonami', 'Thanks for the Telethon library!')
```

The function returns the *custom.Message* that was sent so you can do more things with it if you want.

You can also *reply* or *respond* to messages:

```
message.reply('Hello')
message.respond('World')
```

1.13.8 Sending Markdown or HTML messages

Markdown (`'md'` or `'markdown'`) is the default *parse_mode* for the client. You can change the default parse mode like so:

```
client.parse_mode = 'html'
```

Now all messages will be formatted as HTML by default:

```
client.send_message('me', 'Some <b>bold</b> and <i>italic</i> text')
client.send_message('me', 'An <a href="https://example.com">URL</a>')
client.send_message('me', '<code>code</code> and <pre>pre\nblocks</pre>')
client.send_message('me', '<a href="tg://user?id=me">Mentions</a>')
```

You can override the default parse mode to use for special cases:

```
# No parse mode by default
client.parse_mode = None

# ...but here I want markdown
client.send_message('me', 'Hello, **world**!', parse_mode='md')
```

(continues on next page)

(continued from previous page)

```
# ...and here I need HTML
client.send_message('me', 'Hello, <i>world</i>!', parse_mode='html')
```

The rules are the same as for Bot API, so please refer to https://core.telegram.org/bots/api#formatting-options.

1.13.9 Sending Messages with Media

Sending media can be done with *send_file*:

```
client.send_file(chat, '/my/photos/me.jpg', caption="It's me!")
# or
client.send_message(chat, "It's me!", file='/my/photos/me.jpg')
```

You can send voice notes or round videos by setting the right arguments:

```
client.send_file(chat, '/my/songs/song.mp3', voice_note=True)
client.send_file(chat, '/my/videos/video.mp4', video_note=True)
```

You can set a JPG thumbnail for any document:

```
client.send_file(chat, '/my/documents/doc.txt', thumb='photo.jpg')
```

You can force sending images as documents:

```
client.send_file(chat, '/my/photos/photo.png', force_document=True)
```

You can send albums if you pass more than one file:

```
client.send_file(chat, [
    '/my/photos/holiday1.jpg',
    '/my/photos/holiday2.jpg',
    '/my/drawings/portrait.png'
])
```

The caption can also be a list to match the different photos.

1.13.10 Reusing Uploaded Files

All files you send are automatically cached, so if you do:

```
client.send_file(first_chat, 'document.txt')
client.send_file(second_chat, 'document.txt')
```

The `'document.txt'` file will only be uploaded once. You can disable this behaviour by settings `allow_cache=False`:

```
client.send_file(first_chat, 'document.txt', allow_cache=False)
client.send_file(second_chat, 'document.txt', allow_cache=False)
```

Disabling cache is the only way to send the same document with different attributes (for example, you send an `.ogg` as a song but now you want it to show as a voice note; you probably need to disable the cache).

However, you can *upload* the file once (not sending it yet!), and *then* you can send it with different attributes. This means you can send an image as a photo and a document:

```
file = client.upload_file('photo.jpg')
client.send_file(chat, file)                        # sends as photo
client.send_file(chat, file, force_document=True)   # sends as document

file.name = 'not a photo.jpg'
client.send_file(chat, file, force_document=True)   # document, new name
```

Or, the example described before:

```
file = client.upload_file('song.ogg')
client.send_file(chat, file)                    # sends as song
client.send_file(chat, file, voice_note=True)   # sends as voice note
```

The `file` returned by *client.upload_file* represents the uploaded file, not an immutable document (that's why the attributes can change, because they are set later). This handle can be used only for a limited amount of time (somewhere within a day). Telegram decides this limit and it is not public. However, a day is often more than enough.

1.13.11 Sending Messages with Buttons

You must sign in as a bot in order to add inline buttons (or normal keyboards) to your messages. Once you have signed in as a bot specify the *Button* or buttons to use:

```
from telethon import events
from telethon.tl.custom import Button

@client.on(events.CallbackQuery)
async def callback(event):
    await event.edit('Thank you for clicking {}!'.format(event.data))

client.send_message(chat, 'A single button, with "clk1" as data',
                    buttons=Button.inline('Click me', b'clk1'))

client.send_message(chat, 'Pick one from this grid', buttons=[
    [Button.inline('Left'), Button.inline('Right')],
    [Button.url('Check this site!', 'https://lonamiwebs.github.io')]
])
```

You can also use normal buttons (not inline) to request the user's location, phone number, or simply for them to easily send a message:

```
client.send_message(chat, 'Welcome', buttons=[
    Button.text('Thanks!', resize=True, single_use=True),
    Button.request_phone('Send phone'),
    Button.request_location('Send location')
])
```

Forcing a reply or removing the keyboard can also be done:

```
client.send_message(chat, 'Reply to me', buttons=Button.force_reply())
client.send_message(chat, 'Bye Keyboard!', buttons=Button.clear())
```

Remember to check *Button* for more.

1.13.12 Making Inline Queries

You can send messages `via @bot` by first making an inline query:

```
results = client.inline_query('like', 'Do you like Telethon?')
```

Then access the result you want and *click* it in the chat where you want to send it to:

```
message = results[0].click('TelethonOffTopic')
```

Sending messages through inline bots lets you use buttons as a normal user.

It can look a bit strange at first, but you can make inline queries in no chat in particular, and then click a *result* to send it to some chat.

1.13.13 Clicking Buttons

Let's *click* the message we sent in the example above!

```
message.click(0)
```

This will click the first button in the message. You could also `click(row, column)`, using some text such as `click(text='')` or even the data directly `click(data=b'payload')`.

1.13.14 Answering Inline Queries

As a bot, you can answer to inline queries with *events.InlineQuery*. You should make use of the *builder* property to conveniently build the list of results to show to the user. Remember to check the properties of the *InlineQuery.Event*:

```
@bot.on(events.InlineQuery)
async def handler(event):
    builder = event.builder

    rev_text = event.text[::-1]
    await event.answer([
        builder.article('Reverse text', text=rev_text),
        builder.photo('/path/to/photo.jpg')
    ])
```

1.13.15 Conversations: Waiting for Messages or Replies

This one is really useful for unit testing your bots, which you can even write within Telethon itself! You can open a *Conversation* in any chat as:

```
with client.conversation(chat) as conv:
    ...
```

Conversations let you program a finite state machine with the higher-level constructs we are all used to, such as `while` and `if` conditionals instead setting the state and jumping from one place to another which is less clean.

For instance, let's imagine `you` are the bot talking to `usr`:

```
<you> Hi!
<usr> Hello!
<you> Please tell me your name
<usr> ?
<you> Your name didn't have any letters! Try again
<usr> Lonami
<you> Thanks Lonami!
```

This can be programmed as follows:

```
with bot.conversation(chat) as conv:
    conv.send_message('Hi!')
    hello = conv.get_response()

    conv.send_message('Please tell me your name')
    name = conv.get_response().raw_text
    while not any(x.isalpha() for x in name):
        conv.send_message("Your name didn't have any letters! Try again")
        name = conv.get_response().raw_text

    conv.send_message('Thanks {}!'.format(name))
```

Note how we sent a message **with the conversation**, not with the client. This is important so the conversation remembers what messages you sent.

The method reference for getting a response, getting a reply or marking the conversation as read can be found by clicking here: *Conversation*.

Sending a message or getting a response returns a *Message*. Reading its documentation will also be really useful!

If a reply never arrives or too many messages come in, getting responses will raise `asyncio.TimeoutError` or `ValueError` respectively. You may want to `except` these and tell the user they were too slow, or simply drop the conversation.

1.13.16 Forwarding Messages

You can forward up to 100 messages with *forward_messages*, or a single one if you have the message with *forward_to*:

```
# a single one
client.forward_messages(chat, message)
# or
client.forward_messages(chat, message_id, from_chat)
# or
message.forward_to(chat)

# multiple
client.forward_messages(chat, messages)
# or
client.forward_messages(chat, message_ids, from_chat)
```

You can also "forward" messages without showing "Forwarded from" by re-sending the message:

```
client.send_message(chat, message)
```

1.13.17 Editing Messages

With *edit_message* or *message.edit*:

```
client.edit_message(message, 'New text')
# or
message.edit('New text')
# or
client.edit_message(chat, message_id, 'New text')
```

1.13.18 Deleting Messages

With *delete_messages* or *message.delete*. Note that the first one supports deleting entire chats at once!:

```
client.delete_messages(chat, messages)
# or
message.delete()
```

1.13.19 Marking Messages as Read

Marking messages up to a certain point as read with *send_read_acknowledge*:

```
client.send_read_acknowledge(last_message)
# or
client.send_read_acknowledge(last_message_id)
# or
client.send_read_acknowledge(messages)
```

1.13.20 Getting Entities

Entities are users, chats, or channels. You can get them by their ID if you have seen them before (e.g. you probably need to get all dialogs or all the members from a chat first):

```
from telethon import utils

me = client.get_entity('me')
print(utils.get_display_name(me))

chat = client.get_input_entity('username')
for message in client.iter_messages(chat):
    ...

# Note that you could have used the username directly, but it's
# good to use get_input_entity if you will reuse it a lot.
for message in client.iter_messages('username'):
    ...

# Note that for this to work the phone number must be in your contacts
some_id = client.get_peer_id('+34123456789')
```

The documentation for shown methods are *get_entity*, *get_input_entity* and *get_peer_id*.

Note that the utils package also has a *get_peer_id* but it won't work with things that need access to the network such as usernames or phones, which need to be in your contact list.

1.13.21 Getting the Admin Log

If you're an administrator in a channel or megagroup, then you have access to the admin log. This is a list of events within the last 48 hours of different actions, such as joining or leaving members, edited or deleted messages, new promotions, bans or restrictions.

You can iterate over all the available actions like so:

```
for event in client.iter_admin_log(channel):
    if event.changed_title:
        print('The title changed from', event.old, 'to', event.new)
```

You can also filter to only show some text or actions. Let's find people who swear to ban them:

```
# Get a list of deleted message events which said "heck"
events = client.get_admin_log(channel, search='heck', delete=True)

# Print the old message before it was deleted
print(events[0].old)
```

You can find here the documentation for *client.iter_admin_log*, and be sure to also check the properties of the returned *AdminLogEvent* to know what you can access.

1.14 Working with messages

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

Contents

1.14.1 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
from telethon.tl.functions.messages import GetAllStickersRequest
sticker_sets = client(GetAllStickersRequest(0))

# Choose a sticker set
from telethon.tl.functions.messages import GetStickerSetRequest
from telethon.tl.types import InputStickerSetID
sticker_set = sticker_sets.sets[0]

# Get the stickers for this sticker set
```

(continues on next page)

(continued from previous page)

```
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.send_file('me', stickers.documents[0])
```

1.15 Working with Chats and Channels

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

Contents

1.15.1 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`.

1.15.2 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.

1.15.3 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'))
```

1.15.4 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]
))
```

1.15.5 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.

1.15.6 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 ChatAdminRights

# You need both the channel and who to grant permissions
# They can either be channel/user or input channel/input user.
#
```

(continues on next page)

(continued from previous page)

```
# ChatAdminRights is a list of granted permissions.
# Set to True those you want to give.
rights = ChatAdminRights(
    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 = ChatAdminRights(
#         change_info=True,
#         delete_messages=True,
#         pin_messages=True
#     )

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

# User will now be able to change group info, delete other people's
# messages and pin messages.
#
# In a normal chat, you should do this instead:
from telethon.tl.functions.messages import EditChatAdminRequest

client(EditChatAdminRequest(chat_id, user, is_admin=True))
```

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).

1.15.7 Restricting Users

Similar to how you give or revoke admin permissions, you can edit the banned rights of a user through EditBannedRequest and its parameter ChatBannedRights:

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

from datetime import datetime, timedelta

# Restricting a 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 = ChatBannedRights(
```

(continues on next page)

(continued from previous page)

```
    until_date=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 = ChatBannedRights(
    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))
```

You can also use a `datetime` object for `until_date=`, or even a Unix timestamp. Note that if you ban someone for less than 30 seconds or for more than 366 days, Telegram will consider the ban to actually last forever. This is officially documented under https://core.telegram.org/bots/api#restrictchatmember.

1.15.8 Kicking a member

Telegram doesn't actually have a request to kick a 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 ChatBannedRights

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

1.15.9 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_messages()` or anyhow. Must be a list.

client(GetMessagesViewsRequest(
    peer=channel,
```

(continues on next page)

(continued from previous page)

```
    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.

1.16 Users

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

Contents

1.16.1 Retrieving full information

If you need to retrieve the bio, biography or about information for a user you should use GetFullUser:

```
from telethon.tl.functions.users import GetFullUserRequest

full = client(GetFullUserRequest(user))
# or even
full = client(GetFullUserRequest('username'))

bio = full.about
```

See UserFull to know what other fields you can access.

1.16.2 Updating your name and/or bio

The first name, last name and bio (about) can all be changed with the same request. Omitted fields won't change after invoking UpdateProfile:

```
from telethon.tl.functions.account import UpdateProfileRequest

client(UpdateProfileRequest(
    about='This is a test from Telethon'
))
```

1.16.3 Updating your username

You need to use account.UpdateUsername:

```
from telethon.tl.functions.account import UpdateUsernameRequest

client(UpdateUsernameRequest('new_username'))
```

1.16.4 Updating your profile photo

The easiest way is to upload a new file and use that as the profile photo through UploadProfilePhoto:

```
from telethon.tl.functions.photos import UploadProfilePhotoRequest

client(UploadProfilePhotoRequest(
    client.upload_file('/path/to/some/file')
)))
```

1.17 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!

1.17.1 telethon_examples/

Link / Author's website

This documentation is not the only place where you can find useful code snippets using the library. The main repository also has a folder with some cool examples (even a Tkinter GUI!) which you can download, edit and run to learn and play with them.

1.17.2 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).

1.17.3 mautrix-telegram

Link / Author's website

A Matrix-Telegram hybrid puppeting/relaybot bridge.

1.17.4 TelegramTUI

Link / Author's website

A Telegram client on your terminal.

1.18 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)
```

1.19 Deleted, Limited or Deactivated Accounts

If you're from Iran or Russia, 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.

We have also had reports from Kazakhstan and China, where connecting would fail. To solve these connection problems, you should use a proxy.

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.

1.20 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 attribute). For example:

```
...
from telethon import errors

try:
    print(client.get_messages(chat)[0].text)
except errors.FloodWaitError as e:
    print('Have to sleep', e.seconds, 'seconds')
    time.sleep(e.seconds)
```

- `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`.

You can refer to all errors from Python through the `telethon.errors` module. If you don't know what attributes they have, try printing their dir (like `print(dir(e))`).

1.21 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.

1.22 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'
)
```

1.23 Project Structure

1.23.1 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 from several mixing `Method` classes, since there are so many methods that having them in a single file would make maintenance painful (it was three thousand lines before this separation happened!). It's a "god object", but there is only a way to interact with Telegram really.

The `TelegramBaseClient` is an ABC which will support all of these mixins so they can work together nicely. It doesn't even know how to invoke things because they need to be resolved with user information first (to work with input entities comfortably).

The client makes use of the `network/mtprotosender.py`. The `MTProtoSender` is responsible for connecting, reconnecting, packing, unpacking, sending and receiving items from the network. Basically, the low-level communication with Telegram, and handling MTProto-related functions and types such as `BadSalt`.

The sender makes use of a `Connection` class which knows the format in which outgoing messages should be sent (how to encode their length and their body, if they're further encrypted).

1.23.2 Auto-generated code

The files under `telethon_generator/` are used to generate the code that gets placed under `telethon/tl/`. The parsers take in files in a specific format (such as `.tl` for objects and `.json` for errors) and spit 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.

Custom documentation can also be generated to easily navigate through the vast amount of items offered by the API.

1.24 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`!

1.25 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.

1.26 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!

1.27 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:

1.27.1 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.

1.27.2 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.

1.27.3 JavaScript

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

1.27.4 Kotlin

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

1.27.5 PHP

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

1.27.6 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.

1.27.7 Rust

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

1.28 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

1.28.1 Tidying up Internals (v1.6)

Published at 2019/02/27

Scheme layer used: 95

First things first, sorry for updating the layer in the previous patch version. That should only be done between major versions ideally, but due to how Telegram works, it's done between minor versions. However raw API has and will always be considered "unsafe", this meaning that you should always use the convenience client methods instead. These methods don't cover the full API yet, so pull requests are welcome.

Breaking Changes

- The layer update, of course. This didn't really need a mention here.
- You can no longer pass a `batch_size` when iterating over messages. No other method exposed this parameter, and it was only meant for testing purposes. Instead, it's now a private constant.
- `client.iter_*` methods no longer have a `_total` parameter which was supposed to be private anyway. Instead, they return a new generator object which has a `.total` attribute:

```
it = client.iter_messages(chat)
for i, message in enumerate(it, start=1):
    percentage = i / it.total
    print('{:.2%} {}'.format(percentage, message.text))
```

Additions

- You can now pass `phone` and `phone_code_hash` in *client.sign_up*, although you probably don't need that.
- Thanks to the overhaul of all `client.iter_*` methods, you can now do:

```
for message in reversed(client.iter_messages('me')):
    print(message.text)
```

Bug fixes

- Fix *telethon.utils.resolve_bot_file_id*, which wasn't working after the layer update (so you couldn't send some files by bot file IDs).
- Fix sending albums as bot file IDs (due to image detection improvements).
- Fix *takeout()* failing when they need to download media from other DCs.
- Fix repeatedly calling *conversation.get_response()* when many messages arrived at once (i.e. when several of them were forwarded).
- Fixed connecting with `ConnectionTcpObfuscated`.
- Fix *client.get_peer_id('me')*.
- Fix warning of "missing sqlite3" when in reality it just had wrong tables.
- Fix a strange error when using too many IDs in *client.delete_messages()*.
- Fix *client.send_file* with the result of *client.upload_file*.
- When answering inline results, their order was not being preserved.
- Fix *events.ChatAction* detecting user leaves as if they were kicked.

Enhancements

- Cleared up some parts of the documentation.
- Improved some auto-casts to make life easier.
- Improved image detection. Now you can easily send `bytes` and streams of images as photos, unless you force document.
- Sending images as photos that are too large will now be resized before uploading, reducing the time it takes to upload them and also avoiding errors when the image was too large (as long as `pillow` is installed). The images will remain unchanged if you send it as a document.
- Treat `errors.RpcMcgetFailError` as a temporary server error to automatically retry shortly. This works around most issues.

Internal changes

- New common way to deal with retries (`retry_range`).
- Cleaned up the takeout client.
- Completely overhauled asynchronous generators.

1.28.2 Layer Update (v1.5.5)

Published at 2019/01/14

Scheme layer used: 93

There isn't an entry for v1.5.4 because it contained only one hot-fix regarding loggers. This update is slightly bigger so it deserves mention.

Additions

- New `supports_streaming` parameter in *`client.send_file`*.

Bug fixes

- Dealing with mimetypes should cause less issues in systems like Windows.
- Potentially fix alternative session storages that had issues with dates.

Enhancements

- Saner timeout defaults for conversations.
- `Path`-like files are now supported for thumbnails.
- Added new hot-keys to the online documentation at https://lonamiwebs.github.io/Telethon/ such as `/` to search. Press `?` to view them all.

1.28.3 Bug Fixes (v1.5.3)

Published at 2019/01/14

Several bug fixes and some quality of life enhancements.

Breaking Changes

- *`message.edit`* now respects the previous message buttons or link preview being hidden. If you want to toggle them you need to explicitly set them. This is generally the desired behaviour, but may cause some bots to have buttons when they shouldn't.

Additions

- You can now "hide_via" when clicking on results from *client.inline_query* to @bing and @gif.
- You can now further configure the logger Telethon uses to suit your needs.

Bug fixes

- Fixes for ReadTheDocs to correctly build the documentation.
- Fix UserEmpty not being expected when getting the input variant.
- The message object returned when sending a message with buttons wouldn't always contain the ReplyMarkup.
- Setting email when configuring 2FA wasn't properly supported.
- `utils.resolve_bot_file_id` now works again for photos.

Enhancements

- Chat and channel participants can now be used as peers.
- Reworked README and examples at https://github.com/LonamiWebs/Telethon/tree/master/telethon_examples

1.28.4 Takeout Sessions (v1.5.2)

Published at 2019/01/05

You can now easily start takeout sessions (also known as data export sessions) through *client.takeout()*. Some of the requests will have lower flood limits when done through the takeout session.

Bug fixes

- The new *AdminLogEvent* had a bug that made it unusable.
- *client.iter_dialogs()* will now locally check for the offset date, since Telegram ignores it.
- Answering inline queries with media no works properly. You can now use the library to create inline bots and send stickers through them!

1.28.5 object.to_json() (v1.5.1)

Published at 2019/01/03

The library already had a way to easily convert the objects the API returned into dictionaries through `object.to_dict()`, but some of the fields are dates or `bytes` which JSON can't serialize directly.

For convenience, a new `object.to_json()` has been added which will by default format both of those problematic types into something sensible.

Additions

- New *client.iter_admin_log()* method.

Bug fixes

- *client.is_connected()* would be wrong when the initial connection failed.
- Fixed `UnicodeDecodeError` when accessing the text of messages with malformed offsets in their entities.
- Fixed *client.get_input_entity()* for integer IDs that the client has not seen before.

Enhancements

- You can now configure the reply markup when using *Button* as a bot.
- More properties for *Message* to make accessing media convenient.
- Downloading to `file=bytes` will now return a `bytes` object with the downloaded media.

1.28.6 Polls with the Latest Layer (v1.5)

Published at 2018/12/25

Scheme layer used: 91

This version doesn't really bring many new features, but rather focuses on updating the code base to support the latest available Telegram layer, 91. This layer brings polls, and you can create and manage them through Telethon!

Breaking Changes

- The layer change from 82 to 91 changed a lot of things in the raw API, so be aware that if you rely on raw API calls, you may need to update your code, in particular **if you work with files**. They have a new `file_reference` parameter that you must provide.

Additions

- New *client.is_bot()* method.

Bug fixes

- Markdown and HTML parsing now behave correctly with leading whitespace.
- HTTP connection should now work correctly again.
- Using `caption=None` would raise an error instead of setting no caption.
- `KeyError` is now handled properly when forwarding messages.
- *button.click()* now works as expected for KeyboardButtonGame.

Enhancements

- Some improvements to the search in the full API and generated examples.
- Using entities with `access_hash = 0` will now work in more cases.

Internal changes

- Some changes to the documentation and code generation.
- 2FA code was updated to work under the latest layer.

1.28.7 Error Descriptions in CSV files (v1.4.3)

Published at 2018/12/04

While this may seem like a minor thing, it's a big usability improvement.

Anyone who wants to update the documentation for known errors, or whether some methods can be used as a bot, user or both, can now be easily edited. Everyone is encouraged to help document this better!

Bug fixes

- `TimeoutError` was not handled during automatic reconnects.
- Getting messages by ID using InputMessageReplyTo could fail.
- Fixed *`message.get_reply_message`* as a bot when a user replied to a different bot.
- Accessing some document properties in a *`Message`* would fail.

Enhancements

- Accessing *`events.ChatAction`* properties such as input users may now work in more cases.

Internal changes

- Error descriptions and information about methods is now loaded from a CSV file instead of being part of several messy JSON files.

1.28.8 Bug Fixes (v1.4.2)

Published at 2018/11/24

This version also includes the v1.4.1 hot-fix, which was a single quick fix and didn't really deserve an entry in the changelog.

Bug fixes

- Authorization key wouldn't be saved correctly, requiring re-login.
- Conversations with custom events failed to be cancelled.
- Fixed `telethon.sync` when using other threads.
- Fix markdown/HTML parser from failing with leading/trailing whitespace.
- Fix accessing `chat_action_event.input_user` property.
- Potentially improved handling unexpected disconnections.

Enhancements

- Better default behaviour for *client.send_read_acknowledge*.
- Clarified some points in the documentation.
- Clearer errors for `utils.get_peer*`.

1.28.9 Connection Overhaul (v1.4)

Published at 2018/11/03

Yet again, a lot of work has been put into reworking the low level connection classes. This means `asyncio.open_connection` is now used correctly and the errors it can produce are handled properly. The separation between packing, encrypting and network is now abstracted away properly, so reasoning about the code is easier, making it more maintainable.

As a user, you shouldn't worry about this, other than being aware that quite a few changes were made in the insides of the library and you should report any issues that you encounter with this version if any.

Breaking Changes

- The threaded version of the library will no longer be maintained, primarily because it never was properly maintained anyway. If you have old code, stick with old versions of the library, such as `0.19.1.6`.
- Timeouts no longer accept `timedelta`. Simply use seconds.
- The `callback` parameter from *telethon.tl.custom.button.Button.inline()* was removed, since it had always been a bad idea. Adding the callback there meant a lot of extra work for every message sent, and only registering it after the first message was sent! Instead, use *telethon.events.callbackquery.CallbackQuery*.

Additions

- New *dialog.delete()* method.
- New *conversation.cancel()* method.
- New `retry_delay` delay for the client to be used on auto-reconnection.

Bug fixes

- Fixed *Conversation.wait_event()*.
- Fixed replying with photos/documents on inline results.
- *client.is_user_authorized()* now works correctly after *client.log_out()*.
- *dialog.is_group* now works for ChatForbidden.
- Not using `async with` when needed is now a proper error.
- *events.CallbackQuery* with string regex was not working properly.
- *client.get_entity('me')* now works again.
- Empty codes when signing in are no longer valid.
- Fixed file cache for in-memory sessions.

Enhancements

- Support `next_offset` in *`inline_query.answer()`*.
- Support `<a href="tg://user?id=123">` mentions in HTML parse mode.
- New auto-casts for InputDocument and InputChatPhoto.
- Conversations are now exclusive per-chat by default.
- The request that caused a RPC error is now shown in the error message.
- New full API examples in the generated documentation.
- Fixed some broken links in the documentation.
- *`client.disconnect()`* is now synchronous, but you can still `await` it for consistency or compatibility.

1.28.10 Event Templates (v1.3)

Published at 2018/09/22

If you have worked with Flask templates, you will love this update, since it gives you the same features but even more conveniently:

```
# handlers/welcome.py
from telethon import events

@events.register(events.NewMessage('(?i)hello'))
async def handler(event):
    client = event.client
    await event.respond('Hi!')
    await client.send_message('me', 'Sent hello to someone')
```

This will *`register`* the `handler` callback to handle new message events. Note that you didn't add this to any client yet, and this is the key point: you don't need a client to define handlers! You can add it later:

```
# main.py
from telethon import TelegramClient
import handlers.welcome

with TelegramClient(...) as client:
    # This line adds the handler we defined before for new messages
    client.add_event_handler(handlers.welcome.handler)
    client.run_until_disconnected()
```

This should help you to split your big code base into a more modular design.

Breaking Changes

- `.sender` is the `.chat` when the message is sent in a broadcast channel. This makes sense, because the sender of the message was the channel itself, but you now must take into consideration that it may be either a User or Channel instead of being `None`.

Additions

- New `MultiError` class when invoking many requests at once through `client([requests])`.

- New custom `func=` on all events. These will receive the entire event, and a good usage example is `func=lambda e: e.is_private`.
- New `.web_preview` field on messages. The `.photo` and `.document` will also return the media in the web preview if any, for convenience.
- Callback queries now have a `.chat` in most circumstances.

Bug fixes

- Running code with `python3 -O` would remove critical code from asserts.
- Fix some rare ghost disconnections after reconnecting.
- Fix strange behavior for *`send_message(chat, Message, reply_to=foo)`*.
- The `loop=` argument was being pretty much ignored.
- Fix `MemorySession` file caching.
- The logic for getting entities from their username is now correct.
- Fixes for sending stickers from `.webp` files in Windows, again.
- Fix disconnection without being logged in.
- Retrieving media from messages would fail.
- Getting some messages by ID on private chats.

Enhancements

- *`iter_participants`* will now use its `search=` as a symbol set when `aggressive=True`, so you can do `client.get_participants(group, aggressive=True, search='')`.
- The `StringSession` supports custom encoding.
- Callbacks for *`telethon.client.auth.AuthMethods.start`* can be `async`.

Internal changes

- Cherry-picked a commit to use `asyncio.open_connection` in the lowest level of the library. Do open issues if this causes trouble, but it should otherwise improve performance and reliability.
- Building and resolving events overhaul.

1.28.11 Conversations, String Sessions and More (v1.2)

Published at 2018/08/14

This is a big release! Quite a few things have been added to the library, such as the new *`Conversation`*. This makes it trivial to get tokens from @BotFather:

```
from telethon.tl import types

with client.conversation('BotFather') as conv:
    conv.send_message('/mybots')
    message = conv.get_response()
```

(continues on next page)

(continued from previous page)

```
    message.click(0)
    message = conv.get_edit()
    message.click(0)
    message = conv.get_edit()
    for _, token in message.get_entities_text(types.MessageEntityCode):
        print(token)
```

In addition to that, you can now easily load and export session files without creating any on-disk file thanks to the `StringSession`:

```
from telethon.sessions import StringSession
string = StringSession.save(client.session)
```

Check out *Session Files* for more details.

For those who aren't able to install `cryptg`, the support for `libssl` has been added back. While interfacing `libssl` is not as fast, the speed when downloading and sending files should really be noticeably faster.

While those are the biggest things, there are still more things to be excited about.

Additions

- The mentioned method to start a new *`client.conversation`*.
- Implemented global search through *`client.iter_messages`* with `None` entity.
- New *`client.inline_query`* method to perform inline queries.
- Bot-API-style `file_id` can now be used to send files and download media. You can also access *`telethon.utils.resolve_bot_file_id`* and *`telethon.utils.pack_bot_file_id`* to resolve and create these file IDs yourself. Note that each user has its own ID for each file so you can't use a bot's `file_id` with your user, except stickers.
- New *`telethon.utils.get_peer`*, useful when you expect a Peer.

Bug fixes

- UTC timezone for *`telethon.events.userupdate.UserUpdate`*.
- Bug with certain input parameters when iterating messages.
- RPC errors without parent requests caused a crash, and better logging.
- `incoming = outgoing = True` was not working properly.
- Getting a message's ID was not working.
- File attributes not being inferred for `open()`'ed files.
- Use `MemorySession` if `sqlite3` is not installed by default.
- Self-user would not be saved to the session file after signing in.
- *`client.catch_up()`* seems to be functional again.

Enhancements

- Updated documentation.
- Invite links will now use cache, so using them as entities is cheaper.
- You can reuse message buttons to send new messages with those buttons.
- `.to_dict()` will now work even on invalid `TLObject`'s.

1.28.12 Better Custom Message (v1.1.1)

Published at 2018/07/23

The *`custom.Message`* class has been rewritten in a cleaner way and overall feels less hacky in the library. This should perform better than the previous way in which it was patched.

The release is primarily intended to test this big change, but also fixes **Python 3.5.2 compatibility** which was broken due to a trailing comma.

Bug fixes

- Using `functools.partial` on event handlers broke updates if they had uncaught exceptions.
- A bug under some session files where the sender would export authorization for the same data center, which is unsupported.
- Some logical bugs in the custom message class.

1.28.13 Bot Friendly (v1.1)

Published at 2018/07/21

Two new event handlers to ease creating normal bots with the library, namely *`events.InlineQuery`* and *`events.CallbackQuery`* for handling `@InlineBot queries` or reacting to a button click. For this second option, there is an even better way:

```
from telethon.tl.custom import Button

async def callback(event):
    await event.edit('Thank you!')

bot.send_message(chat, 'Hello!',
                 buttons=Button.inline('Click me', callback))
```

You can directly pass the callback when creating the button.

This is fine for small bots but it will add the callback every time you send a message, so you probably should do this instead once you are done testing:

```
markup = bot.build_reply_markup(Button.inline('Click me', callback))
bot.send_message(chat, 'Hello!', buttons=markup)
```

And yes, you can create more complex button layouts with lists:

```
from telethon import events

global phone = ''

@bot.on(events.CallbackQuery)
async def handler(event):
    global phone
    if event.data == b'<':
        phone = phone[:-1]
    else:
        phone += event.data.decode('utf-8')

    await event.answer('Phone is now {}'.format(phone))

markup = bot.build_reply_markup([
    [Button.inline('1'), Button.inline('2'), Button.inline('3')],
    [Button.inline('4'), Button.inline('5'), Button.inline('6')],
    [Button.inline('7'), Button.inline('8'), Button.inline('9')],
    [Button.inline('+'), Button.inline('0'), Button.inline('<')],
])
bot.send_message(chat, 'Enter a phone', buttons=markup)
```

(Yes, there are better ways to do this). Now for the rest of things:

Additions

- New *custom.Button* class to help you create inline (or normal) reply keyboards. You must sign in as a bot to use the `buttons=` parameters.
- New events usable if you sign in as a bot: *events.InlineQuery* and *events.CallbackQuery*.
- New `silent` parameter when sending messages, usable in broadcast channels.
- Documentation now has an entire section dedicate to how to use the client's friendly methods at *Examples with the Client*.

Bug fixes

- Empty `except` are no longer used which means sending a keyboard interrupt should now work properly.
- The `pts` of incoming updates could be `None`.
- UTC timezone information is properly set for read `datetime`.
- Some infinite recursion bugs in the custom message class.
- Updates was being dispatched to raw handlers when it shouldn't.
- Using proxies and HTTPS connection mode may now work properly.
- Less flood waits when downloading media from different data centers, and the library will now detect them even before sending requests.

Enhancements

- Interactive sign in now supports signing in with a bot token.

- `timedelta` is now supported where a date is expected, which means you can e.g. ban someone for `timedelta(minutes=5)`.
- Events are only built once and reused many times, which should save quite a few CPU cycles if you have a lot of the same type.
- You can now click inline buttons directly if you know their data.

Internal changes

- When downloading media, the right sender is directly used without previously triggering migrate errors.
- Code reusing for getting the chat and the sender, which easily enables this feature for new types.

1.28.14 New HTTP(S) Connection Mode (v1.0.4)

Published at 2018/07/09

This release implements the HTTP connection mode to the library, which means certain proxies that only allow HTTP connections should now work properly. You can use it doing the following, like any other mode:

```
from telethon import TelegramClient, sync
from telethon.network import ConnectionHttp

client = TelegramClient(..., connection=ConnectionHttp)
with client:
    client.send_message('me', 'Hi!')
```

Additions

- `add_mark=` is now back on `utils.get_input_peer` and also on *`client.get_input_entity`*.
- New *`client.get_peer_id`* convenience for `utils.get_peer_id(await client.get_input_entity(peer))`.

Bug fixes

- If several `TLMessage` in a `MessageContainer` exceeds 1MB, it will no longer be automatically turned into one. This basically means that e.g. uploading 10 file parts at once will work properly again.
- Documentation fixes and some missing `await`.
- Revert named argument for *`client.forward_messages`*

Enhancements

- New auto-casts to InputNotifyPeer and `chat_id`.

Internal changes

- Outgoing `TLMessage` are now pre-packed so if there's an error when serializing the raw requests, the library will no longer swallow it. This also means re-sending packets doesn't need to re-pack their bytes.

1.28.15 Iterate Messages in Reverse (v1.0.3)

Published at 2018/07/04

Scheme layer used: 82

Mostly bug fixes, but now there is a new parameter on *client.iter_messages* to support reversing the order in which messages are returned.

Additions

- The mentioned `reverse` parameter when iterating over messages.
- A new `sequential_updates` parameter when creating the client for updates to be processed sequentially. This is useful when you need to make sure that all updates are processed in order, such as a script that only forwards incoming messages somewhere else.

Bug fixes

- Count was always `None` for *message.button_count*.
- Some fixes when disconnecting upon dropping the client.
- Support for Python 3.4 in the sync version, and fix media download.
- Some issues with events when accessing the input chat or their media.
- Hachoir wouldn't automatically close the file after reading its metadata.
- Signing in required a named `code=` parameter, but usage without a name was really widespread so it has been reverted.

1.28.16 Bug Fixes (v1.0.2)

Published at 2018/06/28

Updated some asserts and parallel downloads, as well as some fixes for sync.

1.28.17 Bug Fixes (v1.0.1)

Published at 2018/06/27

And as usual, every major release has a few bugs that make the library unusable! This quick update should fix those, namely:

Bug fixes

- *client.start()* was completely broken due to a last-time change requiring named arguments everywhere.
- Since the rewrite, if your system clock was wrong, the connection would get stuck in an infinite "bad message" loop of responses from Telegram.
- Accessing the buttons of a custom message wouldn't work in channels, which lead to fix a completely different bug regarding starting bots.

- Disconnecting could complain if the magic `telethon.sync` was imported.
- Successful automatic reconnections now ask Telegram to send updates to us once again as soon as the library is ready to listen for them.

1.28.18 Synchronous magic (v1.0)

Published at 2018/06/27

Important: If you come from Telethon pre-1.0 you **really** want to read *Compatibility and Convenience* to port your scripts to the new version.

The library has been around for well over a year. A lot of improvements have been made, a lot of user complaints have been fixed, and a lot of user desires have been implemented. It's time to consider the public API as stable, and remove some of the old methods that were around until now for compatibility reasons. But there's one more surprise!

There is a new magic `telethon.sync` module to let you use **all** the methods in the *TelegramClient* (and the types returned from its functions) in a synchronous way, while using `asyncio` behind the scenes! This means you're now able to do both of the following:

```
import asyncio

async def main():
  await client.send_message('me', 'Hello!')

asyncio.get_event_loop().run_until_complete(main())

# ...can be rewritten as:

from telethon import sync
client.send_message('me', 'Hello!')
```

Both ways can coexist (you need to `await` if the loop is running).

You can also use the magic `sync` module in your own classes, and call `sync.syncify(cls)` to convert all their `async def` into magic variants.

Breaking Changes

- `message.get_fwd_sender` is now in *`message.forward`*.
- `client.idle` is now *`client.run_until_disconnected()`*
- `client.add_update_handler` is now *`client.add_event_handler`*
- `client.remove_update_handler` is now *`client.remove_event_handler`*
- `client.list_update_handlers` is now *`client.list_event_handlers`*
- `client.get_message_history` is now *`client.get_messages`*
- `client.send_voice_note` is now *`client.send_file`* with `is_voice=True`.
- `client.invoke()` is now `client(...)`.
- `report_errors` has been removed since it's currently not used, and `flood_sleep_threshold` is now part of the client.

- The `update_workers` and `spawn_read_thread` arguments are gone. Simply remove them from your code when you create the client.
- Methods with a lot of arguments can no longer be used without specifying their argument. Instead you need to use named arguments. This improves readability and not needing to learn the order of the arguments, which can also change.

Additions

- *`client.send_file`* now accepts external `http://` and `https://` URLs.
- You can use the *TelegramClient* inside of `with` blocks, which will *`client.start()`* and *`disconnect()`* the client for you:

```
from telethon import TelegramClient, sync

with TelegramClient(name, api_id, api_hash) as client:
    client.send_message('me', 'Hello!')
```

Convenience at its maximum! You can even chain the *`.start()`* method since it returns the instance of the client:

```
with TelegramClient(name, api_id, api_hash).start(bot_token=token) as bot:
    bot.send_message(chat, 'Hello!')
```

Bug fixes

- There were some `@property async def` left, and some `await property`.
- "User joined" event was being treated as "User was invited".
- SQLite's cursor should not be closed properly after usage.
- `await` the updates task upon disconnection.
- Some bug in Python 3.5.2's `asyncio` causing 100% CPU load if you forgot to call *`client.disconnect()`*. The method is called for you on object destruction, but you still should disconnect manually or use a `with` block.
- Some fixes regarding disconnecting on client deletion and properly saving the authorization key.
- Passing a class to *`message.get_entities_text`* now works properly.
- Iterating messages from a specific user in private messages now works.

Enhancements

- Both *`client.start()`* and *`client.run_until_disconnected()`* can be ran in both a synchronous way (without starting the loop manually) or from an `async def` where they need to have an `await`.

1.28.19 Core Rewrite in asyncio (v1.0-rc1)

Published at 2018/06/24

Scheme layer used: 81

This version is a major overhaul of the library internals. The core has been rewritten, cleaned up and refactored to fix some oddities that have been growing inside the library.

This means that the code is easier to understand and reason about, including the code flow such as conditions, exceptions, where to reconnect, how the library should behave, and separating different retry types such as disconnections or call fails, but it also means that **some things will necessarily break** in this version.

All requests that touch the network are now methods and need to have their `await` (or be ran until their completion).

Also, the library finally has the simple logo it deserved: a carefully hand-written `.svg` file representing a T following Python's colours.

Breaking Changes

- If you relied on internals like the `MtProtoSender` and the `TelegramBareClient`, both are gone. They are now *`MTProtoSender`* and *`TelegramBaseClient`* and they behave differently.
- Underscores have been renamed from filenames. This means `telethon.errors.rpc_error_list` won't work, but you should have been using `telethon.errors` all this time instead.
- *`client.connect`* no longer returns `True` on success. Instead, you should `except` the possible `ConnectionError` and act accordingly. This makes it easier to not ignore the error.
- You can no longer set `retries=n` when calling a request manually. The limit works differently now, and it's done on a per-client basis.
- Accessing `.sender`, `.chat` and similar may *not* work in events anymore, since previously they could access the network. The new rule is that properties are not allowed to make API calls. You should use `.get_sender()`, `.get_chat()` instead while using events. You can safely access properties if you get messages through *`client.get_messages()`* or other methods in the client.
- The above point means `reply_message` is now *`.get_reply_message()`*, and `fwd_from_entity` is now `get_fwd_sender()`. Also `forward` was gone in the previous version, and you should be using `fwd_from` instead.

Additions

- Telegram's Terms Of Service are now accepted when creating a new account. This can possibly help avoid bans. This has no effect for accounts that were created before.
- The method reference now shows which methods can be used if you sign in with a `bot_token`.
- There's a new *`client.disconnected`* future which you can wait on. When a disconnection occurs, you will now, instead letting it happen in the background.
- More configurable retries parameters, such as auto-reconnection, retries when connecting, and retries when sending a request.
- You can filter *`events.NewMessage`* by sender ID, and also whether they are forwards or not.
- New `ignore_migrated` parameter for *`client.iter_dialogs`*.

Bug fixes

- Several fixes to *`telethon.events.newmessage.NewMessage`*.
- Removed named `length` argument in `to_bytes` for PyPy.
- Raw events failed due to not having `._set_client`.

- *message.get_entities_text* properly supports filtering, even if there are no message entities.
- *message.click* works better.
- The server started sending DraftMessageEmpty which the library didn't handle correctly when getting dialogs.
- The "correct" chat is now always returned from returned messages.
- `to_id` was not validated when retrieving messages by their IDs.
- `'__'` is no longer considered valid in usernames.
- The `fd` is removed from the reader upon closing the socket. This should be noticeable in Windows.
- MessageEmpty is now handled when searching messages.
- Fixed a rare infinite loop bug in *client.iter_dialogs* for some people.
- Fixed `TypeError` when there is no `.sender`.

Enhancements

- You can now delete over 100 messages at once with *client.delete_messages*.
- Signing in now accounts for `AuthRestartError` itself, and also handles `PasswordHashInvalidError`.
- `__all__` is now defined, so `from telethon import *` imports sane defaults (client, events and utils). This is however discouraged and should be used only in quick scripts.
- `pathlib.Path` is now supported for downloading and uploading media.
- Messages you send to yourself are now considered outgoing, unless they are forwarded.
- The documentation has been updated with a brand new `asyncio` crash course to encourage you use it. You can still use the threaded version if you want though.
- `.name` property is now properly supported when sending and downloading files.
- Custom `parse_mode`, which can now be set per-client, support MessageEntityMentionName so you can return those now.
- The session file is saved less often, which could result in a noticeable speed-up when working with a lot of incoming updates.

Internal changes

- The flow for sending a request is as follows: the `TelegramClient` creates a `MTProtoSender` with a `Connection`, and the sender starts send and receive loops. Sending a request means enqueueing it in the sender, which will eventually pack and encrypt it with its `ConnectionState` instead of using the entire `Session` instance. When the data is packed, it will be sent over the `Connection` and ultimately over the `TcpClient`.
- Reconnection occurs at the `MTProtoSender` level, and receiving responses follows a similar process, but now `asyncio.Future` is used for the results which are no longer part of all `TLObject`, instead are part of the `TLMessage` which simplifies things.
- Objects can no longer be `content_related` and instead subclass `TLRequest`, making the separation of concerns easier.
- The `TelegramClient` has been split into several mixin classes to avoid having a 3,000-lines-long file with all the methods.

- More special cases in the `MTProtoSender` have been cleaned up, and also some attributes from the `Session` which didn't really belong there since they weren't being saved.
- The `telethon_generator/` can now convert `.tl` files into `.json`, mostly as a proof of concept, but it might be useful for other people.

1.28.20 Custom Message class (v0.19.1)

Published at 2018/06/03

Scheme layer used: 80

This update brings a new *`telethon.tl.custom.message.Message`* object!

All the methods in the `telethon.telegram_client.TelegramClient` that used to return a Message will now return this object instead, which means you can do things like the following:

```
msg = client.send_message(chat, 'Hello!')
msg.edit('Hello there!')
msg.reply('Good day!')
print(msg.sender)
```

Refer to its documentation to see all you can do, again, click *`telethon.tl.custom.message.Message`* to go to its page.

Breaking Changes

- The `telethon.network.connection.common.Connection` class is now an ABC, and the old `ConnectionMode` is now gone. Use a specific connection (like `telethon.network.connection.tcpabridged.ConnectionTcpAbridged`) instead.

Additions

- You can get messages by their ID with `telethon.telegram_client.TelegramClient.get_messages`'s `ids` parameter:

```
message = client.get_messages(chats, ids=123)  # Single message
message_list = client.get_messages(chats, ids=[777, 778])  # Multiple
```

- More convenience properties for *`telethon.tl.custom.dialog.Dialog`*.
- New default `telethon.telegram_client.TelegramClient.parse_mode`.
- You can edit the media of messages that already have some media.
- New dark theme in the online `tl` reference, check it out at https://lonamiwebs.github.io/Telethon/.

Bug fixes

- Some IDs start with `1000` and these would be wrongly treated as channels.
- Some short usernames like `@vote` were being ignored.
- `telethon.telegram_client.TelegramClient.iter_messages`'s `from_user` was failing if no filter had been set.

- `telethon.telegram_client.TelegramClient.iter_messages`'s `min_id/max_id` was being ignored by Telegram. This is now worked around.
- `telethon.telegram_client.TelegramClient.catch_up` would fail with empty states.
- *`telethon.events.newmessage.NewMessage`* supports `incoming=False` to indicate `outgoing=True`.

Enhancements

- You can now send multiple requests at once while preserving the order:

```
from telethon.tl.functions.messages import SendMessageRequest
client([SendMessageRequest(chat, 'Hello 1!'),
        SendMessageRequest(chat, 'Hello 2!')], ordered=True)
```

Internal changes

- `without rowid` is not used in SQLite anymore.
- Unboxed serialization would fail.
- Different default limit for `iter_messages` and `get_messages`.
- Some clean-up in the `telethon_generator/` package.

1.28.21 Catching up on Updates (v0.19)

Published at 2018/05/07

Scheme layer used: 76

This update prepares the library for catching up with updates with the new `telethon.telegram_client.TelegramClient.catch_up` method. This feature needs more testing, but for now it will let you "catch up" on some old updates that occurred while the library was offline, and brings some new features and bug fixes.

Additions

- Add `search`, `filter` and `from_user` parameters to `telethon.telegram_client.TelegramClient.iter_messages`.
- `telethon.telegram_client.TelegramClient.download_file` now supports a `None` path to return the file in memory and return its `bytes`.
- Events now have a `.original_update` field.

Bug fixes

- Fixed a race condition when receiving items from the network.
- A disconnection is made when "retries reached 0". This hasn't been tested but it might fix the bug.
- `reply_to` would not override Message object's reply value.
- Add missing caption when sending Message with media.

Enhancements

- Retry automatically on `RpcCallFailError`. This error happened a lot when iterating over many messages, and retrying often fixes it.
- Faster `telethon.telegram_client.TelegramClient.iter_messages` by sleeping only as much as needed.
- `telethon.telegram_client.TelegramClient.edit_message` now supports omitting the entity if you pass a Message.
- *`telethon.events.raw.Raw`* can now be filtered by type.

Internal changes

- The library now distinguishes between MTProto and API schemas.
- State is now persisted to the session file.
- Connection won't retry forever.
- Fixed some errors and cleaned up the generation of code.
- Fixed typos and enhanced some documentation in general.
- Add auto-cast for InputMessage and InputLocation.

1.28.22 Pickle-able objects (v0.18.3)

Published at 2018/04/15

Now you can use Python's `pickle` module to serialize `RPCError` and any other `TLObject` thanks to **@vegeta1k95**! A fix that was fairly simple, but still might be useful for many people.

As a side note, the documentation at https://lonamiwebs.github.io/Telethon now lists known `RPCError` for all requests, so you know what to expect. This required a major rewrite, but it was well worth it!

Breaking changes

- `telethon.telegram_client.TelegramClient.forward_messages` now returns a single item instead of a list if the input was also a single item.

Additions

- New *`telethon.events.messageread.MessageRead`* event, to find out when and who read which messages as soon as it happens.
- Now you can access `.chat_id` on all events and `.sender_id` on some.

Bug fixes

- Possibly fix some bug regarding lost `GzipPacked` requests.
- The library now uses the "real" layer 75, hopefully.
- Fixed `.entities` name collision on updates by making it private.

- AUTH_KEY_DUPLICATED is handled automatically on connection.
- Markdown parser's offset uses match.start() to allow custom regex.
- Some filter types (as a type) were not supported by telethon.telegram_client.TelegramClient.iter_participants.
- telethon.telegram_client.TelegramClient.remove_event_handler works.
- telethon.telegram_client.TelegramClient.start works on all terminals.
- InputPeerSelf case was missing from telethon.telegram_client.TelegramClient.get_input_entity.

Enhancements

- The parse_mode for messages now accepts a callable.
- telethon.telegram_client.TelegramClient.download_media accepts web previews.
- *telethon.tl.custom.dialog.Dialog* instances can now be casted into InputPeer.
- Better logging when reading packages "breaks".
- Better and more powerful setup.py gen command.

Internal changes

- The library won't call .get_dialogs() on entity not found. Instead, it will raise ValueError() so you can properly except it.
- Several new examples and updated documentation.
- py:obj is the default Sphinx's role which simplifies .rst files.
- setup.py now makes use of python_requires.
- Events now live in separate files.
- Other minor changes.

1.28.23 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.

1.28.24 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.

1.28.25 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.

1.28.26 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.

1.28.27 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.

1.28.28 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.

1.28.29 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.

1.28.30 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).

1.28.31 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.

1.28.32 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).

1.28.33 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!

1.28.34 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.

1.28.35 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.

1.28.36 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)`.

1.28.37 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.

1.28.38 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.

1.28.39 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!

1.28.40 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.

1.28.41 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.

1.28.42 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.

1.28.43 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`).

1.28.44 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.

1.28.45 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.

1.28.46 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.

1.28.47 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.

1.28.48 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.

1.28.49 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.

1.28.50 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!

1.28.51 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 .

1.28.52 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!

1.28.53 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.

1.28.54 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).

1.28.55 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.

1.28.56 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.

1.28.57 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.

1.28.58 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()`.

1.28.59 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.

1.28.60 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.

1.28.61 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.

1.28.62 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=`.

1.28.63 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.

1.28.64 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!

1.28.65 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.

1.28.66 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!

1.28.67 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**.

1.28.68 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.

1.28.69 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.

1.28.70 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!

1.28.71 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.

1.29 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.

1.29.1 Current winner

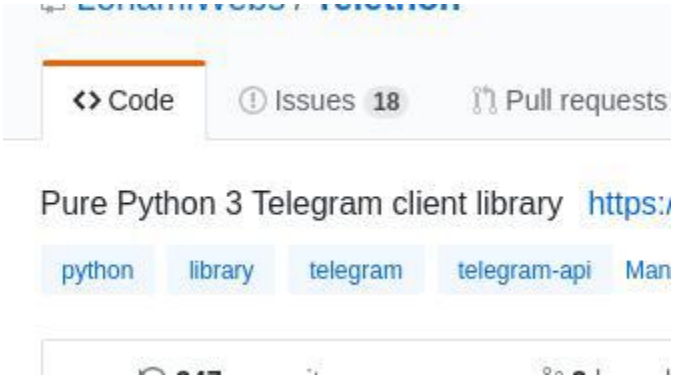
The current winner is issue 213:

Issue:

alt Winner issue

Winner issue

Answer:



alt Winner issue answer

Winner issue answer

1.30 telethon

1.30.1 telethon package

telethon.client module

telethon.client package

The *telethon.TelegramClient* aggregates several mixin classes to provide all the common functionality in a nice, Pythonic interface. Each mixin has its own methods, which you all can use.

In short, to create a client you must run:

```
import asyncio
from telethon import TelegramClient

async def main():
    client = await TelegramClient(name, api_id, api_hash).start()
    # Now you can use all client methods listed below, like for example...
    await client.send_message('me', 'Hello to myself!')

asyncio.get_event_loop().run_until_complete(main())
```

You **don't** need to import these `AuthMethods`, `MessageMethods`, etc. Together they are the *telethon.TelegramClient* and you can access all of their methods.

class telethon.client.telegrambaseclient.**TelegramBaseClient**(*session*, *api_id*, *api_hash*, ***, *connection=<class 'telethon.network.connection.tcpfull.Connecti*, *use_ipv6=False*, *proxy=None*, *timeout=10*, *request_retries=5*, *connection_retries=5*, *retry_delay=1*, *auto_reconnect=True*, *sequential_updates=False*, *flood_sleep_threshold=60*, *device_model=None*, *system_version=None*, *app_version=None*, *lang_code='en'*, *system_lang_code='en'*, *loop=None*, *base_logger=None*)

Bases: abc.ABC

This is the abstract base class for the client. It defines some basic stuff like connecting, switching data center, etc, and leaves the __call__ unimplemented.

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.

Note that if you pass a string it will be a file in the current working directory, although you can also pass absolute paths.

The session file contains enough information for you to login without re-sending the code, so if you have to enter the code more than once, maybe you're changing the working directory, renaming or removing the file, or using random names.

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 (telethon.network.connection.common.Connection, optional): The connection instance to be used when creating a new connection to the servers. It **must** be a type.

Defaults to telethon.network.connection.tcpfull.ConnectionTcpFull.

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 | list | dict, optional): An iterable consisting of the proxy info. If connection is one of MTProxy, then it should contain MTProxy credentials: ('hostname', port, 'secret'). Otherwise, it's meant to store function parameters for PySocks, like (type, 'hostname', port). See https://github.com/Anorov/PySocks#usage-1 for more.

timeout (int | float, optional): The timeout in seconds to be used when connecting. This is **not** the timeout to be used when `await`'ing for invoked requests, and you should use `asyncio.wait` or `asyncio.wait_for` for that.

request_retries (int | None, optional): How many times a request should be retried. Request are retried when Telegram is having internal issues (due to either `errors.ServerError` or `errors.RpcCallFailError`), when there is a `errors.FloodWaitError` less than `flood_sleep_threshold`, or when there's a migrate error.

May take a negative or `None` value for infinite retries, but this is not recommended, since some requests can always trigger a call fail (such as searching for messages).

connection_retries (int | None, optional): How many times the reconnection should retry, either on the initial connection or when Telegram disconnects us. May be set to a negative or `None` value for infinite retries, but this is not recommended, since the program can get stuck in an infinite loop.

retry_delay (int | float, optional): The delay in seconds to sleep between automatic reconnections.

auto_reconnect (bool, optional): Whether reconnection should be retried `connection_retries` times automatically if Telegram disconnects us or not.

sequential_updates (bool, optional): By default every incoming update will create a new task, so you can handle several updates in parallel. Some scripts need the order in which updates are processed to be sequential, and this setting allows them to do so.

If set to `True`, incoming updates will be put in a queue and processed sequentially. This means your event handlers should *not* perform long-running operations since new updates are put inside of an unbounded queue.

flood_sleep_threshold (int | float, optional): The threshold below which the library should automatically sleep on flood wait errors (inclusive). For instance, if a `FloodWaitError` for 17s occurs and `flood_sleep_threshold` is 20s, the library will `sleep` automatically. If the error was for 21s, it would `raise FloodWaitError` instead. Values larger than a day (like `float('inf')`) will be changed to a day.

device_model (str, optional): "Device model" to be sent when creating the initial connection. Defaults to `platform.node()`.

system_version (str, optional): "System version" to be sent when creating the initial connection. Defaults to `platform.system()`.

app_version (str, optional): "App version" to be sent when creating the initial connection. Defaults to `telethon.version.__version__`.

lang_code (str, optional): "Language code" to be sent when creating the initial connection. Defaults to `'en'`.

system_lang_code (str, optional): "System lang code" to be sent when creating the initial connection. Defaults to `lang_code`.

loop (asyncio.AbstractEventLoop, optional): Asyncio event loop to use. Defaults to `asyncio.get_event_loop()`

base_logger (str | logging.Logger, optional): Base logger name or instance to use. If a `str` is given, it'll be passed to `logging.getLogger()`. If a `logging.Logger` is given, it'll be used directly. If something else or nothing is given, the default logger will be used.

connect()
Connects to Telegram.

disconnect()
Disconnects from Telegram.

If the event loop is already running, this method returns a coroutine that you should await on your own code; otherwise the loop is ran until said coroutine completes.

disconnected
Future that resolves when the connection to Telegram ends, either by user action or in the background.

is_connected()
Returns `True` if the user has connected.

loop

class telethon.client.account.**AccountMethods**(*session*, *api_id*, *api_hash*, *, *connection=<class 'telethon.network.connection.tcpfull.ConnectionTcpFull'>*, *use_ipv6=False*, *proxy=None*, *timeout=10*, *request_retries=5*, *connection_retries=5*, *retry_delay=1*, *auto_reconnect=True*, *sequential_updates=False*, *flood_sleep_threshold=60*, *device_model=None*, *system_version=None*, *app_version=None*, *lang_code='en'*, *system_lang_code='en'*, *loop=None*, *base_logger=None*)

Bases: *telethon.client.users.UserMethods*

end_takeout(*success*)
Finishes a takeout, with specified result sent back to Telegram.

Returns: `True` if the operation was successful, `False` otherwise.

takeout(*finalize=True*, *, *contacts=None*, *users=None*, *chats=None*, *megagroups=None*, *channels=None*, *files=None*, *max_file_size=None*)
Creates a proxy object over the current TelegramClient through which making requests will use InvokeWithTakeoutRequest to wrap them. In other words, returns the current client modified so that requests are done as a takeout:

```
>>> from telethon.sync import TelegramClient
>>>
>>> with TelegramClient(...) as client:
>>>     with client.takeout() as takeout:
>>>         client.get_messages('me')   # normal call
>>>         takeout.get_messages('me')  # wrapped through takeout
```

Some of the calls made through the takeout session will have lower flood limits. This is useful if you want to export the data from conversations or mass-download media, since the rate limits will be lower. Only some requests will be affected, and you will need to adjust the `wait_time` of methods like *client.iter_messages*.

By default, all parameters are `None`, and you need to enable those you plan to use by setting them to either `True` or `False`.

You should `except errors.TakeoutInitDelayError as e`, since this exception will raise depending on the condition of the session. You can then access `e.seconds` to know how long you should wait for before calling the method again.

There's also a `success` property available in the takeout proxy object, so from the `with` body you can set the boolean result that will be sent back to Telegram. But if it's left `None` as by default, then the action is based on the `finalize` parameter. If it's `True` then the takeout will be finished, and if no exception occurred during it, then `True` will be considered as a result. Otherwise, the takeout will not be finished and its ID will be preserved for future usage as `client.session.takeout_id`.

Args:

contacts (bool): Set to `True` if you plan on downloading contacts.

users (bool): Set to `True` if you plan on downloading information from users and their private conversations with you.

chats (bool): Set to `True` if you plan on downloading information from small group chats, such as messages and media.

megagroups (bool): Set to `True` if you plan on downloading information from megagroups (channels), such as messages and media.

channels (bool): Set to `True` if you plan on downloading information from broadcast channels, such as messages and media.

files (bool): Set to `True` if you plan on downloading media and you don't only wish to export messages.

max_file_size (int): The maximum file size, in bytes, that you plan to download for each message with media.

class telethon.client.auth.**AuthMethods**(*session*, *api_id*, *api_hash*, *, *connection=<class 'telethon.network.connection.tcpfull.ConnectionTcpFull'>*, *use_ipv6=False*, *proxy=None*, *timeout=10*, *request_retries=5*, *connection_retries=5*, *retry_delay=1*, *auto_reconnect=True*, *sequential_updates=False*, *flood_sleep_threshold=60*, *device_model=None*, *system_version=None*, *app_version=None*, *lang_code='en'*, *system_lang_code='en'*, *loop=None*, *base_logger=None*)

Bases: *telethon.client.messageparse.MessageParseMethods*, *telethon.client.users.UserMethods*

edit_2fa(*current_password=None*, *new_password=None*, *, *hint=''*, *email=None*, *email_code_callback=None*)

Changes the 2FA settings of the logged in user, according to the passed parameters. Take note of the parameter explanations.

Note that this method may be *incredibly* slow depending on the prime numbers that must be used during the process to make sure that everything is safe.

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. If present, you must also set `email_code_callback`, else it raises `ValueError`.

email_code_callback (callable, optional): If an email is provided, a callback that returns the code sent to it must also be set. This callback may be asynchronous. It should return a string with the code. The length of the code will be passed to the callback as an input parameter.

If the callback returns an invalid code, it will raise `CodeInvalidError`.

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

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

Returns: `True` if the operation was successful.

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.

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, optional): The hash returned by *send_code_request*. This can be left as `None` to use the last hash known for the phone to be used.

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

sign_up(*code*, *first_name*, *last_name=''*, *, *phone=None*, *phone_code_hash=None*)
Signs up to Telegram if you don't have an account yet. You must call .send_code_request(phone) first.

By using this method you're agreeing to Telegram's Terms of Service. This is required and your account will be banned otherwise. See https://telegram.org/tos and https://core.telegram.org/api/terms.

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.

phone (str | int, optional): The phone to sign up. This will be the last phone used by default (you normally don't need to set this).

phone_code_hash (str, optional): The hash returned by *send_code_request*. This can be left as `None` to use the last hash known for the phone to be used.

Returns: The new created User.

start(*phone=<function AuthMethods.<lambda>>*, *password=<function AuthMethods.<lambda>>*, ***, *bot_token=None*, *force_sms=False*, *code_callback=None*, *first_name='New User'*, *last_name=''*, *max_attempts=3*)

Convenience method to interactively connect and sign in if required, also taking into consideration that 2FA may be enabled in the account.

If the phone doesn't belong to an existing account (and will hence `sign_up` for a new one), **you are agreeing to Telegram's Terms of Service. This is required and your account will be banned otherwise.** See https://telegram.org/tos and https://core.telegram.org/api/terms.

Example usage:

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

If the event loop is already running, this method returns a coroutine that you should await on your own code; otherwise the loop is ran until said coroutine completes.

Args:

phone (`str` | `int` | `callable`): The phone (or callable without arguments to get it) to which the code will be sent. If a bot-token-like string is given, it will be used as such instead. The argument may be a coroutine.

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

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()`. The argument may be a coroutine.

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.

max_attempts (`int`, optional): How many times the code/password callback should be retried or switching between signing in and signing up.

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

class telethon.client.bots.**BotMethods**(*session*, *api_id*, *api_hash*, ***, *connection=<class 'telethon.network.connection.tcpfull.ConnectionTcpFull'>*, *use_ipv6=False*, *proxy=None*, *timeout=10*, *request_retries=5*, *connection_retries=5*, *retry_delay=1*, *auto_reconnect=True*, *sequential_updates=False*, *flood_sleep_threshold=60*, *device_model=None*, *system_version=None*, *app_version=None*, *lang_code='en'*, *system_lang_code='en'*, *loop=None*, *base_logger=None*)

Bases: *telethon.client.users.UserMethods*

inline_query(*bot*, *query*, *, *offset=None*, *geo_point=None*)
Makes the given inline query to the specified bot i.e. `@vote My New Poll` would be as follows:

```
>>> client = ...
>>> client.inline_query('vote', 'My New Poll')
```

Args:

bot (entity): The bot entity to which the inline query should be made.

query (str): The query that should be made to the bot.

offset (str, optional): The string offset to use for the bot.

geo_point (GeoPoint, optional) The geo point location information to send to the bot for localised results. Available under some bots.

Returns: A list of *custom.InlineResult*.

class telethon.client.buttons.**ButtonMethods**(*session*, *api_id*, *api_hash*, *, *connection=<class 'telethon.network.connection.tcpfull.ConnectionTcpFull'>*, *use_ipv6=False*, *proxy=None*, *timeout=10*, *request_retries=5*, *connection_retries=5*, *retry_delay=1*, *auto_reconnect=True*, *sequential_updates=False*, *flood_sleep_threshold=60*, *device_model=None*, *system_version=None*, *app_version=None*, *lang_code='en'*, *system_lang_code='en'*, *loop=None*, *base_logger=None*)
Bases: *telethon.client.updates.UpdateMethods*

build_reply_markup(*buttons*, *inline_only=False*)
Builds a :tl‘ReplyInlineMarkup‘ or ReplyKeyboardMarkup for the given buttons, or does nothing if either no buttons are provided or the provided argument is already a reply markup.

This will add any event handlers defined in the buttons and delete old ones not to call them twice, so you should probably call this method manually for serious bots instead re-adding handlers every time you send a message. Magic can only go so far.

class telethon.client.chats.**ChatMethods**(*session*, *api_id*, *api_hash*, *, *connection=<class 'telethon.network.connection.tcpfull.ConnectionTcpFull'>*, *use_ipv6=False*, *proxy=None*, *timeout=10*, *request_retries=5*, *connection_retries=5*, *retry_delay=1*, *auto_reconnect=True*, *sequential_updates=False*, *flood_sleep_threshold=60*, *device_model=None*, *system_version=None*, *app_version=None*, *lang_code='en'*, *system_lang_code='en'*, *loop=None*, *base_logger=None*)
Bases: *telethon.client.users.UserMethods*

get_admin_log(**args*, ***kwargs*)
Same as *iter_admin_log*, but returns a `list` instead.

get_participants(**args*, ***kwargs*)
Same as *iter_participants*, but returns a *TotalList* instead.

iter_admin_log(*entity*, *limit=None*, ***, *max_id=0*, *min_id=0*, *search=None*, *admins=None*, *join=None*, *leave=None*, *invite=None*, *restrict=None*, *unrestrict=None*, *ban=None*, *unban=None*, *promote=None*, *demote=None*, *info=None*, *settings=None*, *pinned=None*, *edit=None*, *delete=None*)

Iterator over the admin log for the specified channel.

Note that you must be an administrator of it to use this method.

If none of the filters are present (i.e. they all are `None`), *all* event types will be returned. If at least one of them is `True`, only those that are true will be returned.

Args:

entity (`entity`): The channel entity from which to get its admin log.

limit (`int` | `None`, optional): Number of events to be retrieved.

The limit may also be `None`, which would eventually return the whole history.

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

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

search (`str`): The string to be used as a search query.

admins (`entity` | `list`): If present, the events will be filtered by these admins (or single admin) and only those caused by them will be returned.

join (`bool`): If `True`, events for when a user joined will be returned.

leave (`bool`): If `True`, events for when a user leaves will be returned.

invite (`bool`): If `True`, events for when a user joins through an invite link will be returned.

restrict (`bool`): If `True`, events with partial restrictions will be returned. This is what the API calls "ban".

unrestrict (`bool`): If `True`, events removing restrictions will be returned. This is what the API calls "unban".

ban (`bool`): If `True`, events applying or removing all restrictions will be returned. This is what the API calls "kick" (restricting all permissions removed is a ban, which kicks the user).

unban (`bool`): If `True`, events removing all restrictions will be returned. This is what the API calls "unkick".

promote (`bool`): If `True`, events with admin promotions will be returned.

demote (`bool`): If `True`, events with admin demotions will be returned.

info (`bool`): If `True`, events changing the group info will be returned.

settings (`bool`): If `True`, events changing the group settings will be returned.

pinned (`bool`): If `True`, events of new pinned messages will be returned.

edit (`bool`): If `True`, events of message edits will be returned.

delete (`bool`): If `True`, events of message deletions will be returned.

Yields: Instances of *telethon.tl.custom.adminlogevent.AdminLogEvent*.

iter_participants(*entity*, *limit=None*, ***, *search=''*, *filter=None*, *aggressive=False*)

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.

If `aggressive is True`, the symbols from this string will be used.

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.

Note: The filter ChannelParticipantsBanned will return *restricted* users. If you want *banned* users you should use ChannelParticipantsKicked instead.

---

aggressive (`bool`, optional): Aggressively looks for all participants in the chat.

This is useful for channels since 20 July 2018, Telegram added a server-side limit where only the first 200 members can be retrieved. With this flag set, more than 200 will be often be retrieved.

This has no effect if a `filter` is given.

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.

class telethon.client.dialogs.**DialogMethods**(*session*, *api_id*, *api_hash*, *, *connection=<class 'telethon.network.connection.tcpfull.ConnectionTcpFull'>*, *use_ipv6=False*, *proxy=None*, *timeout=10*, *request_retries=5*, *connection_retries=5*, *retry_delay=1*, *auto_reconnect=True*, *sequential_updates=False*, *flood_sleep_threshold=60*, *device_model=None*, *system_version=None*, *app_version=None*, *lang_code='en'*, *system_lang_code='en'*, *loop=None*, *base_logger=None*)

Bases: *telethon.client.users.UserMethods*

conversation(*entity*, *, *timeout=60*, *total_timeout=None*, *max_messages=100*, *exclusive=True*, *replies_are_responses=True*)

Creates a *Conversation* with the given entity so you can easily send messages and await for responses or other reactions. Refer to its documentation for more.

Args:

entity (`entity`): The entity with which a new conversation should be opened.

timeout (`int` | `float`, optional): The default timeout (in seconds) *per action* to be used. You may also override this timeout on a per-method basis. By default each action can take up to 60 seconds (the value of this timeout).

total_timeout (`int` | `float`, optional): The total timeout (in seconds) to use for the whole conversation. This takes priority over per-action timeouts. After these many seconds pass, subsequent actions will result in `asyncio.TimeoutError`.

max_messages (`int`, optional): The maximum amount of messages this conversation will remember. After these many messages arrive in the specified chat, subsequent actions will result in `ValueError`.

exclusive (bool, optional): By default, conversations are exclusive within a single chat. That means that while a conversation is open in a chat, you can't open another one in the same chat, unless you disable this flag.

If you try opening an exclusive conversation for a chat where it's already open, it will raise `AlreadyInConversationError`.

replies_are_responses (bool, optional): Whether replies should be treated as responses or not.

If the setting is enabled, calls to *conv.get_response* and a subsequent call to *conv.get_reply* will return different messages, otherwise they may return the same message.

Consider the following scenario with one outgoing message, 1, and two incoming messages, the second one replying:

```
                    Hello! <1
2> (reply to 1) Hi!
3> (reply to 1) How are you?
```

And the following code:

```
async with client.conversation(chat) as conv:
    msg1 = await conv.send_message('Hello!')
    msg2 = await conv.get_response()
    msg3 = await conv.get_reply()
```

With the setting enabled, `msg2` will be `'Hi!'` and `msg3` be `'How are you?'` since replies are also responses, and a response was already returned.

With the setting disabled, both `msg2` and `msg3` will be `'Hi!'` since one is a response and also a reply.

Returns: A *Conversation*.

get_dialogs(**args*, ***kwargs*)
Same as *iter_dialogs*, but returns a *TotalList* instead.

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

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

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.

ignore_migrated (bool, optional): Whether Chat that have `migrated_to` a Channel should be included or not. By default all the chats in your dialogs are returned, but setting this to `True` will hide them in the same way official applications do.

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.

class telethon.client.downloads.**DownloadMethods**(*session*, *api_id*, *api_hash*, *, *connection=<class 'telethon.network.connection.tcpfull.ConnectionTcpFull'>*, *use_ipv6=False*, *proxy=None*, *timeout=10*, *request_retries=5*, *connection_retries=5*, *retry_delay=1*, *auto_reconnect=True*, *sequential_updates=False*, *flood_sleep_threshold=60*, *device_model=None*, *system_version=None*, *app_version=None*, *lang_code='en'*, *system_lang_code='en'*, *loop=None*, *base_logger=None*)

Bases: *telethon.client.users.UserMethods*

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

Args:

input_location (FileLocation | InputFileLocation): The file location from which the file will be downloaded. See *telethon.utils.get_input_location* source for a complete list of supported types.

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.

If the file path is `None` or `bytes`, then the result will be saved in memory and returned as `bytes`.

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.

Note that if the download is too slow, you should consider installing `cryptg` (through `pip install cryptg`) so that decrypting the received data is done in C instead of Python (much faster).

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. If file is the type `bytes`, it will be downloaded in-memory as a bytestring (e.g. `file=bytes`).

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.

Note: This method expects the full entity (which has the data to download the photo), not an input variant.

It's possible that sometimes you can't fetch the entity from its input (since you can get errors like `ChannelPrivateError`) but you already have it through another call, like getting a forwarded message from it.

---

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. If file is the type `bytes`, it will be downloaded in-memory as a bytestring (e.g. `file=bytes`).

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.

class telethon.client.messageparse.**MessageParseMethods**(*session*, *api_id*, *api_hash*, *, *connection=<class 'telethon.network.connection.tcpfull.ConnectionTcpF*, *use_ipv6=False*, *proxy=None*, *timeout=10*, *request_retries=5*, *connection_retries=5*, *retry_delay=1*, *auto_reconnect=True*, *sequential_updates=False*, *flood_sleep_threshold=60*, *device_model=None*, *system_version=None*, *app_version=None*, *lang_code='en'*, *system_lang_code='en'*, *loop=None*, *base_logger=None*)

Bases: *telethon.client.users.UserMethods*

parse_mode
This property is the default parse mode used when sending messages. Defaults to *telethon.extensions.markdown*. It will always be either `None` or an object with `parse` and `unparse` methods.

When setting a different value it should be one of:

- Object with `parse` and `unparse` methods.
- A `callable` to act as the parse method.
- A `str` indicating the `parse_mode`. For Markdown `'md'` or `'markdown'` may be used. For HTML, `'htm'` or `'html'` may be used.

The `parse` method should be a function accepting a single parameter, the text to parse, and returning a tuple consisting of `(parsed message str, [MessageEntity instances])`.

The `unparse` method should be the inverse of `parse` such that `assert text == unparse(*parse(text))`.

See MessageEntity for allowed message entities.

class telethon.client.messages.**MessageMethods**(*session*, *api_id*, *api_hash*, *, *connection=<class 'telethon.network.connection.tcpfull.ConnectionTcpFull'>*, *use_ipv6=False*, *proxy=None*, *timeout=10*, *request_retries=5*, *connection_retries=5*, *retry_delay=1*, *auto_reconnect=True*, *sequential_updates=False*, *flood_sleep_threshold=60*, *device_model=None*, *system_version=None*, *app_version=None*, *lang_code='en'*, *system_lang_code='en'*, *loop=None*, *base_logger=None*)

Bases: *telethon.client.uploads.UploadMethods*, *telethon.client.buttons.ButtonMethods*, *telethon.client.messageparse.MessageParseMethods*

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: A list of AffectedMessages, each item being the result for the delete calls of the messages in chunks of 100 each.

edit_message(*entity*, *message=None*, *text=None*, *, *parse_mode=()*, *link_preview=True*, *file=None*, *buttons=None*)

Edits the given message ID (to change its contents or disable preview).

Args:

entity (`entity` | *`Message`*): From which chat to edit the message. This can also be the message to be edited, and the entity will be inferred from it, so the next parameter will be assumed to be the message text.

You may also pass a InputBotInlineMessageID, which is the only way to edit messages that were sent after the user selects an inline query result.

message (`int` | *`Message`* | `str`): The ID of the message (or *`Message`* itself) to be edited. If the `entity` was a *`Message`*, then this message will be treated as the new text.

text (`str`, optional): The new text of the message. Does nothing if the `entity` was a *`Message`*.

parse_mode (`object`, optional): See the *`TelegramClient.parse_mode`* property for allowed values. Markdown parsing will be used by default.

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

file (`str | bytes | file | media`, optional): The file object that should replace the existing media in the message.

buttons (`list`, *`custom.Button`*, KeyboardButton): The matrix (list of lists), row list or button to be shown after sending the message. This parameter will only work if you have signed in as a bot. You can also pass your own ReplyMarkup here.

Examples:

```
>>> client = ...
>>> message = client.send_message('username', 'hello')
>>>
>>> client.edit_message('username', message, 'hello!')
>>> # or
>>> client.edit_message('username', message.id, 'Hello')
>>> # or
>>> client.edit_message(message, 'Hello!')
```

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

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

Returns: The edited *`telethon.tl.custom.message.Message`*, unless `entity` was a InputBotInlineMessageID in which case this method returns a boolean.

`forward_messages`(*entity*, *messages*, *from_peer=None*, *, *silent=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. This parameter indicates the entity from which the messages should be forwarded.

silent (`bool`, optional): Whether the message should notify people in a broadcast channel or not. Defaults to `False`, which means it will notify them. Set it to `True` to alter this behaviour.

Returns: The list of forwarded *`telethon.tl.custom.message.Message`*, or a single one if a list wasn't provided as input.

Note that if all messages are invalid (i.e. deleted) the call will fail with `MessageIdInvalidError`. If only some are invalid, the list will have `None` instead of those messages.

`get_messages`(**args*, ***kwargs*)

Same as *`iter_messages`*, but returns a *`TotalList`* instead.

If the `limit` is not set, it will be 1 by default unless both `min_id` **and** `max_id` are set (as *named* arguments), in which case the entire range will be returned.

This is so because any integer limit would be rather arbitrary and it's common to only want to fetch one message, but if a range is specified it makes sense that it should return the entirety of it.

If `ids` is present in the *named* arguments and is not a list, a single *`Message`* will be returned for convenience instead of a list.

iter_messages(*entity*, *limit=None*, ***, *offset_date=None*, *offset_id=0*, *max_id=0*, *min_id=0*, *add_offset=0*, *search=None*, *filter=None*, *from_user=None*, *wait_time=None*, *ids=None*, *reverse=False*)

Iterator over the message history for the specified entity. If either `search`, `filter` or `from_user` are provided, messages.Search will be used instead of messages.getHistory.

Args:

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

It may be `None` to perform a global search, or to get messages by their ID from no particular chat. Note that some of the offsets will not work if this is the case.

Note that if you want to perform a global search, you **must** set a non-empty `search` string.

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).

search (`str`): The string to be used as a search query.

filter (MessagesFilter | `type`): The filter to use when returning messages. For instance, InputMessagesFilterPhotos would yield only messages containing photos.

from_user (`entity`): Only messages from this user will be returned. This parameter will be ignored if it is not an user.

wait_time (`int`): Wait time (in seconds) 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.

ids (`int`, `list`): A single integer ID (or several IDs) for the message that should be returned. This parameter takes precedence over the rest (which will be ignored if this is set). This can for instance be used to get the message with ID 123 from a channel. Note that if the message doesn't exist, `None` will appear in its place, so that zipping the list of IDs with the messages can match one-to-one.

Note: At the time of writing, Telegram will **not** return MessageEmpty for InputMessageReplyTo IDs that failed (i.e. the message is not replying to any, or is replying to a deleted message). This means that it is **not** possible to match messages one-by-one, so be careful if you use non-integers in this parameter.

---

reverse (`bool`, optional): If set to `True`, the messages will be returned in reverse order (from oldest to newest, instead of the default newest to oldest). This also means that the meaning of `offset_id` and `offset_date` parameters is reversed, although they will still be exclusive. `min_id` becomes equivalent to `offset_id` instead of being `max_id` as well since messages are returned in ascending order.

You cannot use this if both `entity` and `ids` are `None`.

Yields: Instances of *telethon.tl.custom.message.Message*.

Notes: Telegram's flood wait limit for GetHistoryRequest seems to be around 30 seconds per 10 requests, therefore a sleep of 1 second is the default for this limit (or above).

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

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.

Sending a `/start` command with a parameter (like `?start=data`) is also done through this method. Simply send `'/start data'` to the bot.

Args:

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

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

The maximum length for a message is 35,000 bytes or 4,096 characters. Longer messages will not be sliced automatically, and you should slice them manually if the text to send is longer than said length.

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 (object, optional): See the *TelegramClient.parse_mode* property for allowed values. Markdown parsing will be used by default.

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.

buttons (list, *custom.Button*, KeyboardButton): The matrix (list of lists), row list or button to be shown after sending the message. This parameter will only work if you have signed in as a bot. You can also pass your own ReplyMarkup here.

All the following limits apply together:

- There can be 100 buttons at most (any more are ignored).
- There can be 8 buttons per row at most (more are ignored).
- The maximum callback data per button is 64 bytes.
- The maximum data that can be embedded in total is just over 4KB, shared between inline callback data and text.

silent (bool, optional): Whether the message should notify people in a broadcast channel or not. Defaults to `False`, which means it will notify them. Set it to `True` to alter this behaviour.

Returns: The sent *custom.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").

This effectively marks a message as read (or more than one) in the given conversation.

If neither message nor maximum ID are provided, all messages will be marked as read by assuming that `max_id = 0`.

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.

class telethon.client.updates.**EventBuilderDict**(*client*, *update*)
Bases: `object`

Helper "dictionary" to return events from types and cache them.

class telethon.client.updates.**UpdateMethods**(*session*, *api_id*, *api_hash*, *, *connection=<class 'telethon.network.connection.tcpfull.ConnectionTcpFull'>*, *use_ipv6=False*, *proxy=None*, *timeout=10*, *request_retries=5*, *connection_retries=5*, *retry_delay=1*, *auto_reconnect=True*, *sequential_updates=False*, *flood_sleep_threshold=60*, *device_model=None*, *system_version=None*, *app_version=None*, *lang_code='en'*, *system_lang_code='en'*, *loop=None*, *base_logger=None*)
Bases: *telethon.client.users.UserMethods*

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.

Note that if you have used *telethon.events.register* in the callback, `event` will be ignored, and instead the events you previously registered will be used.

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

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

catch_up()
"Catches up" on the missed updates while the client was offline. You should call this method after registering the event handlers so that the updates it loads can by processed by your script.

This can also be used to forcibly fetch new updates if there are any.

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

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

```
>>> from telethon import TelegramClient, events
>>> client = TelegramClient(...)
>>>
>>> @client.on(events.NewMessage)
... async def handler(event):
...     ...
...
>>>
```

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.

run_until_disconnected()
Runs the event loop until `disconnect` is called or if an error while connecting/sending/receiving occurs in the background. In the latter case, said error will `raise` so you have a chance to `except` it on your own code.

If the loop is already running, this method returns a coroutine that you should await on your own code.

class telethon.client.uploads.**UploadMethods**(*session*, *api_id*, *api_hash*, *, *connection=<class 'telethon.network.connection.tcpfull.ConnectionTcpFull'>*, *use_ipv6=False*, *proxy=None*, *timeout=10*, *request_retries=5*, *connection_retries=5*, *retry_delay=1*, *auto_reconnect=True*, *sequential_updates=False*, *flood_sleep_threshold=60*, *device_model=None*, *system_version=None*, *app_version=None*, *lang_code='en'*, *system_lang_code='en'*, *loop=None*, *base_logger=None*)
Bases: *telethon.client.buttons.ButtonMethods*, *telethon.client.messageparse.MessageParseMethods*, *telethon.client.users.UserMethods*

send_file(*entity*, *file*, *, *caption=None*, *force_document=False*, *progress_callback=None*, *reply_to=None*, *attributes=None*, *thumb=None*, *allow_cache=True*, *parse_mode=()*, *voice_note=False*, *video_note=False*, *buttons=None*, *silent=None*, *supports_streaming=False*, ***kwargs*)
Sends a file to the specified entity.

Args:

entity (entity): Who will receive the file.

file (str | bytes | file | media): The file to send, which can be one of:

- A local file path to an in-disk file. The file name will be the path's base name.

- A `bytes` byte array with the file's data to send (for example, by using `text.encode('utf-8')`). A default file name will be used.
- A bytes `io.IOBase` stream over the file to send (for example, by using `open(file, 'rb')`). Its `.name` property will be used for the file name, or a default if it doesn't have one.
- An external URL to a file over the internet. This will send the file as "external" media, and Telegram is the one that will fetch the media and send it.
- A Bot API-like `file_id`. You can convert previously sent media to file IDs for later reusing with *telethon.utils.pack_bot_file_id*.
- A handle to an existing file (for example, if you sent a message with media before, you can use its `message.media` as a file here).
- A handle to an uploaded file (from *upload_file*).

To send an album, you should provide a list in this parameter.

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. When sending an album, the caption may be a list of strings, which will be assigned to the files pairwise.

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 JPEG thumbnail (for documents). **Telegram will ignore this parameter** unless you pass a `.jpg` file!

The file must also be small in dimensions and in-disk size. Successful thumbnails were files below 20kb and 200x200px. Width/height and dimensions/size ratios may be important.

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 (`object`, optional): See the *TelegramClient.parse_mode* property for allowed values. Markdown parsing will be used by default.

voice_note (`bool`, optional): If `True` the audio will be sent as a voice note.

Set `allow_cache` to `False` if you sent the same file without this setting before for it to work.

video_note (`bool`, optional): If `True` the video will be sent as a video note, also known as a round video message.

Set `allow_cache` to `False` if you sent the same file without this setting before for it to work.

buttons (`list`, *custom.Button*, KeyboardButton): The matrix (list of lists), row list or button to be shown after sending the message. This parameter will only work if you have signed in as a bot. You can also pass your own ReplyMarkup here.

silent (`bool`, optional): Whether the message should notify people in a broadcast channel or not. Defaults to `False`, which means it will notify them. Set it to `True` to alter this behaviour.

supports_streaming (bool, optional): Whether the sent video supports streaming or not. Note that Telegram only recognizes as streamable some formats like MP4, and others like AVI or MKV will not work. You should convert these to MP4 before sending if you want them to be streamable. Unsupported formats will result in `VideoContentTypeError`.

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

If the `pillow` package is installed and you are sending a photo, it will be resized to fit within the maximum dimensions allowed by Telegram to avoid `errors.PhotoInvalidDimensionsError`. This cannot be done if you are sending InputFile, however.

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

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".

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, `telethon.tl.custom.inputsizedfile.InputSizedFile` (subclass of InputFile) otherwise.

class telethon.client.users.**UserMethods**(*session*, *api_id*, *api_hash*, ***, *connection=<class 'telethon.network.connection.tcpfull.ConnectionTcpFull'>*, *use_ipv6=False*, *proxy=None*, *timeout=10*, *request_retries=5*, *connection_retries=5*, *retry_delay=1*, *auto_reconnect=True*, *sequential_updates=False*, *flood_sleep_threshold=60*, *device_model=None*, *system_version=None*, *app_version=None*, *lang_code='en'*, *system_lang_code='en'*, *loop=None*, *base_logger=None*)

Bases: *telethon.client.telegrambaseclient.TelegramBaseClient*

get_entity(*entity*)

Turns the given entity into a valid Telegram User, Chat or Channel. You can also pass a list or iterable of entities, and they will be efficiently fetched from the network.

entity (`str` | `int` | Peer | InputPeer): If a username is given, **the username will be resolved** making an API call every time. Resolving usernames is an expensive operation and will start hitting flood waits around 50 usernames in a short period of time.

If you want to get the entity for a *cached* username, you should first *get_input_entity(username)* which will use the cache), and then use *get_entity* with the result of the previous call.

Similar limits apply to invite links, and you should use their ID instead.

Using phone numbers (from people in your contact list), exact names, integer IDs or Peer rely on a *get_input_entity* first, which in turn needs the entity to be in cache, unless a InputPeer was passed.

Unsupported types will raise `TypeError`.

If the entity can't be found, `ValueError` 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 InputPeer, so this is the most suitable call to make for those cases. **Generally you should let the library do its job** and don't worry about getting the input entity first, but if you're going to use an entity often, consider making the call:

```
>>> import asyncio
>>> rc = asyncio.get_event_loop().run_until_complete
>>>
>>> from telethon import TelegramClient
>>> client = TelegramClient(...)
>>> # If you're going to use "username" often in your code
>>> # (make a lot of calls), consider getting its input entity
>>> # once, and then using the "user" everywhere instead.
>>> user = rc(client.get_input_entity('username'))
>>> # The same applies to IDs, chats or channels.
>>> chat = rc(client.get_input_entity(-123456789))
```

entity (`str` | `int` | Peer | InputPeer): If a username or invite link is given, **the library will use the cache**. This means that it's possible to be using a username that *changed* or an old invite link (this only happens if an invite link for a small group chat is used after it was upgraded to a mega-group).

If the username or ID from the invite link is not found in the cache, it will be fetched. The same rules apply to phone numbers (`'+34 123456789'`) from people in your contact list.

If an exact name is given, it must be in the cache too. This is not reliable as different people can share the same name and which entity is returned is arbitrary, and should be used only for quick tests.

If a positive integer ID is given, the entity will be searched in cached users, chats or channels, without making any call.

If a negative integer ID is given, the entity will be searched exactly as either a chat (prefixed with −) or as a channel (prefixed with `-100`).

If a Peer is given, it will be searched exactly in the cache as either a user, chat or channel.

If the given object can be turned into an input entity directly, said operation will be done.

Unsupported types will raise `TypeError`.

If the entity can't be found, `ValueError` will be raised.

Returns: InputPeerUser, InputPeerChat or InputPeerChannel or InputPeerSelf if the parameter is `'me'` or `'self'`.

If you need to get the ID of yourself, you should use *get_me* with `input_peer=True`) instead.

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_peer_id(*peer*, *add_mark=True*)
Gets the ID for the given peer, which may be anything entity-like.

This method needs to be `async` because `peer` supports usernames, invite-links, phone numbers (from people in your contact list), etc.

If `add_mark is False`, then a positive ID will be returned instead. By default, bot-API style IDs (signed) are returned.

is_bot()
Return `True` if the signed-in user is a bot, `False` otherwise.

is_user_authorized()
Returns `True` if the user is authorized.

This package defines clients as subclasses of others, and then a single `telethon.client.telegramclient.TelegramClient` which is subclass of them all to provide the final unified interface while the methods can live in different subclasses to be more maintainable.

The ABC is *telethon.client.telegrambaseclient.TelegramBaseClient* and the first implementor is *telethon.client.users.UserMethods*, since calling requests require them to be resolved first, and that requires accessing entities (users).

telethon.utils module

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

class telethon.utils.**AsyncClassWrapper**(*wrapped*)
Bases: `object`

telethon.utils.**chunks**(*iterable*, *size=100*)
Turns the given iterable into chunks of the specified size, which is 100 by default since that's what Telegram uses the most.

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_attributes**(*file*, *, *attributes=None*, *mime_type=None*, *force_document=False*, *voice_note=False*, *video_note=False*, *supports_streaming=False*)
Get a list of attributes for the given file and the mime type as a tuple ([attribute], mime_type).

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_inner_text**(*text*, *entities*)
Gets the inner text that's surrounded by the given entities. For instance: text = 'hey!', entity = MessageEntityBold(2, 2) -> 'y!'.

Parameters

- **text** – the original text.
- **entities** – the entity or entities that must be matched.

Returns a single result or a list of the text surrounded by the entities.

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

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

telethon.utils.**get_input_dialog**(*dialog*)
Similar to *get_input_peer()*, but for dialogs

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_location**(*location*)
Similar to *get_input_peer()*, but for input messages.

Note that this returns a tuple `(dc_id, location)`, the `dc_id` being present if known.

telethon.utils.**get_input_media**(*media*, *, *is_photo=False*, *attributes=None*, *force_document=False*, *voice_note=False*, *video_note=False*, *supports_streaming=False*)
Similar to *get_input_peer()*, but for media.

If the media is InputFile and `is_photo` is known to be `True`, it will be treated as an InputMediaUploadedPhoto. Else, the rest of parameters will indicate how to treat it.

telethon.utils.**get_input_message**(*message*)
Similar to *get_input_peer()*, but for input messages.

telethon.utils.**get_input_peer**(*entity*, *allow_self=True*, *check_hash=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 or if `check_hash is True` but the entity's `access_hash is None`.

Note that `check_hash` **is ignored** if an input peer is already passed since in that case we assume the user knows what they're doing. This is key to getting entities by explicitly passing `hash = 0`.

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_message_id**(*message*)
Similar to *get_input_peer()*, but for message IDs.

telethon.utils.**get_peer**(*peer*)

telethon.utils.**get_peer_id**(*peer*, *add_mark=True*)
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_gif**(*file*)
Returns `True` if the file extension looks like a gif file to Telegram.

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.**pack_bot_file_id**(*file*)
Inverse operation for *resolve_bot_file_id*.

The only parameters this method will accept are Document and Photo, and it will return a variable-length `file_id` string.

If an invalid parameter is given, it will `return None`.

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, False)` if the `username` or link is not valid.

telethon.utils.**resolve_bot_file_id**(*file_id*)
Given a Bot API-style `file_id`, returns the media it represents. If the `file_id` is not valid, `None` is returned instead.

Note that the `file_id` does not have information such as image dimensions or file size, so these will be zero if present.

For thumbnails, the photo ID and hash will always be zero.

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

telethon.utils.**resolve_inline_message_id**(*inline_msg_id*)
Resolves an inline message ID. Returns a tuple of `(message id, peer, dc id, access hash)`

The `peer` may either be a PeerUser referencing the user who sent the message via the bot in a private conversation or small group chat, or a PeerChannel if the message was sent in a channel.

The `access_hash` does not have any use yet.

telethon.utils.**resolve_invite_link**(*link*)
Resolves the given invite link. Returns a tuple of `(link creator user id, global chat id, random int)`.

Note that for broadcast channels, the link creator user ID will be zero to protect their identity. Normal chats and megagroup channels will have such ID.

Note that the chat ID may not be accurate for chats with a link that were upgraded to megagroup, since the link can remain the same, but the chat ID will be correct once a new link is generated.

telethon.utils.**sanitize_parse_mode**(*mode*)
Converts the given parse mode into an object with `parse` and `unparse` callable properties.

telethon.helpers module

Various helpers not related to the Telegram API itself

class telethon.helpers.**TotalList**(**args*, ***kwargs*)
Bases: `list`

A list with an extra `total` property, which may not match its `len` since the total represents the total amount of items *available* somewhere else, not the items *in this list*.

telethon.helpers.**add_surrogate**(*text*)

telethon.helpers.**del_surrogate**(*text*)

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.**retry_range**(*retries*)
Generates an integer sequence starting from 1. If `retries` is not a zero or a positive integer value, the sequence will be infinite, otherwise it will end at `retries + 1`.

telethon.helpers.**strip_text**(*text*, *entities*)
Strips whitespace from the given text modifying the provided entities.

This assumes that there are no overlapping entities, that their length is greater or equal to one, and that their length is not out of bounds.

telethon.events package

telethon.events package

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*, *func=None*)
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.), preferably IDs. 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`.

func (callable, optional): A callable function that should accept the event as input parameter, and return a value indicating whether the event should be dispatched or not (any truthy value will do, it does not need to be a `bool`). It works like a custom filter:

```
@client.on(events.NewMessage(func=lambda e: e.is_private))
async def handler(event):
    pass  # code here
```

classmethod build(*update*)
Builds an event for the given update if possible, or returns None

filter(*event*)
If the ID of `event._chat_peer` isn't in the chats set (or it is but the set is a blacklist) returns `None`, otherwise the event.

The events must have been resolved before this can be called.

resolve(*client*)
Helper method to allow event builders to be resolved before usage

self_id = None

class telethon.events.common.**EventCommon**(*chat_peer=None*, *msg_id=None*, *broadcast=False*)
Bases: *telethon.tl.custom.chatgetter.ChatGetter*, abc.ABC

Intermediate class with common things to all events.

Remember that this class implements *ChatGetter* which means you have access to all chat properties and methods.

In addition, you can access the `original_update` field which contains the original Update.

client
The *telethon.TelegramClient* that created this event.

stringify()

to_dict()

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

class telethon.events.newmessage.**NewMessage**(*chats=None*, ***, *blacklist_chats=False*, *func=None*, *incoming=None*, *outgoing=None*, *from_users=None*, *forwards=None*, *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).

from_users (entity, optional): Unlike `chats`, this parameter filters the *senders* of the message. That is, only messages *sent by these users* will be handled. Use `chats` if you want private messages with this/these users. `from_users` lets you filter by messages sent by *one or more* users across the desired chats (doesn't need a list).

forwards (bool, optional): Whether forwarded messages should be handled or not. By default, both forwarded and normal messages are included. If it's `True` *only* forwards will be handled. If it's `False` only messages that are *not* forwards will be handled.

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. This event can be treated to all effects as a *telethon.tl.custom.message.Message*, so please **refer to its documentation** to know what you can do with this event.

Members:

message (*Message*): This is the only difference with the received *telethon.tl.custom.message.Message*, and will return the *telethon.tl.custom.message.Message* itself, not the text.

See *telethon.tl.custom.message.Message* for the rest of available members and methods.

pattern_match (obj): The resulting object from calling the passed `pattern` function. Here's an example using a string (defaults to regex match):

```
>>> from telethon import TelegramClient, events
>>> client = TelegramClient(...)
>>>
>>> @client.on(events.NewMessage(pattern=r'hi (\w+)!'))
... async def handler(event):
...     # In this case, the result is a ``Match`` object
...     # since the ``str`` pattern was converted into
...     # the ``re.compile(pattern).match`` function.
...     print('Welcomed', event.pattern_match.group(1))
...
>>>
```

classmethod build(*update*)

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

filter(*event*)

If the ID of `event._chat_peer` isn't in the chats set (or it is but the set is a blacklist) returns `None`, otherwise the event.

The events must have been resolved before this can be called.

class telethon.events.chataction.**ChatAction**(*chats=None*, ***, *blacklist_chats=False*, *func=None*)

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 (str, 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. Shorthand for *telethon.client.messages.MessageMethods.delete_messages* with `entity` and `message_ids` already set.

Does nothing if no message action triggered this event.

get_added_by()
Returns *added_by* but will make an API call if necessary.

get_input_user()
Returns *input_user* but will make an API call if necessary.

get_input_users()
Returns *input_users* but will make an API call if necessary.

get_kicked_by()
Returns *kicked_by* but will make an API call if necessary.

get_pinned_message()
If `new_pin` is `True`, this returns the *telethon.tl.custom.message.Message* object that was pinned.

get_user()
Returns *user* but will make an API call if necessary.

get_users()
Returns *users* but will make an API call if necessary.

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).

reply(**args*, ***kwargs*)
Replies to the chat action message (as a reply). Shorthand for *telethon.client.messages.MessageMethods.send_message* with both `entity` and `reply_to` already set.

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 *telethon.client.messages.MessageMethods.send_message* with `entity` already set.

user
The first 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.

user_id
Returns the marked signed ID of the first user, if any.

user_ids
Returns the marked signed ID of the users, if any.

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.

classmethod build(*update*)
Builds an event for the given update if possible, or returns None

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

Represents a user update (gone online, offline, joined Telegram).

class Event(*user_id*, ***, *status=None*, *typing=None*)
Bases: *telethon.events.common.EventCommon*

Represents the event of a 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.

get_input_user()
Alias for `get_input_chat`.

get_user()
Alias for `get_chat` (conversation).

input_user
Alias for `input_chat`.

user
Alias for `chat` (conversation).

user_id
Alias for `chat_id`.

classmethod build(*update*)
Builds an event for the given update if possible, or returns None

class telethon.events.messageedited.**MessageEdited**(*chats=None*, ***, *blacklist_chats=False*, *func=None*, *incoming=None*, *outgoing=None*, *from_users=None*, *forwards=None*, *pattern=None*)
Bases: *telethon.events.newmessage.NewMessage*

Event fired when a message has been edited.

class Event(*message*)
Bases: telethon.events.newmessage.Event

classmethod build(*update*)
Builds an event for the given update if possible, or returns None

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

Event fired when one or more messages are deleted.

class Event(*deleted_ids*, *peer*)
Bases: *telethon.events.common.EventCommon*

classmethod build(*update*)
Builds an event for the given update if possible, or returns None

class telethon.events.messageread.**MessageRead**(*chats=None*, ***, *blacklist_chats=None*, *func=None*, *inbox=False*)
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.

get_messages()
Returns the list of *telethon.tl.custom.message.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.

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.

classmethod build(*update*)
Builds an event for the given update if possible, or returns None

filter(*event*)
If the ID of `event._chat_peer` isn't in the chats set (or it is but the set is a blacklist) returns `None`, otherwise the event.

The events must have been resolved before this can be called.

class telethon.events.callbackquery.**CallbackQuery**(*chats=None*, ***, *blacklist_chats=False*, *func=None*, *data=None*)
Bases: *telethon.events.common.EventBuilder*

Represents a callback query event (when an inline button is clicked).

Note that the `chats` parameter will **not** work with normal IDs or peers if the clicked inline button comes from a "via bot" message. The `chats` parameter also supports checking against the `chat_instance` which should be used for inline callbacks.

Args:

data (bytes | str | callable, optional): If set, the inline button payload data must match this data. A UTF-8 string can also be given, a regex or a callable. For instance, to check against `'data_1'` and `'data_2'` you can use `re.compile(b'data_')`.

class Event(*query*, *peer*, *msg_id*)

Bases: *telethon.events.common.EventCommon*, *telethon.tl.custom.sendergetter.SenderGetter*

Represents the event of a new callback query.

Members:

query (UpdateBotCallbackQuery): The original UpdateBotCallbackQuery.

data_match (obj, optional): The object returned by the `data=` parameter when creating the event builder, if any. Similar to `pattern_match` for the new message event.

answer(*message=None*, *cache_time=0*, ***, *url=None*, *alert=False*)

Answers the callback query (and stops the loading circle).

Args:

message (str, optional): The toast message to show feedback to the user.

cache_time (int, optional): For how long this result should be cached on the user's client. Defaults to 0 for no cache.

url (str, optional): The URL to be opened in the user's client. Note that the only valid URLs are those of games your bot has, or alternatively a 't.me/your_bot?start=xyz' parameter.

alert (bool, optional): Whether an alert (a pop-up dialog) should be used instead of showing a toast. Defaults to `False`.

chat_instance

Unique identifier for the chat where the callback occurred. Useful for high scores in games.

data

Returns the data payload from the original inline button.

delete(**args*, ***kwargs*)

Deletes the message. Shorthand for *telethon.client.messages.MessageMethods.delete_messages* with `entity` and `message_ids` already set.

If you need to delete more than one message at once, don't use this *delete* method. Use a `telethon.client.telegramclient.TelegramClient` instance directly.

This method also creates a task to *answer* the callback.

This method will likely fail if *via_inline* is `True`.

edit(**args*, ***kwargs*)

Edits the message. Shorthand for *telethon.client.messages.MessageMethods.edit_message* with the `entity` set to the correct InputBotInlineMessageID.

Returns `True` if the edit was successful.

This method also creates a task to *answer* the callback.

Note: This method won't respect the previous message unlike *Message.edit*, since the message object is normally not present.

---

get_message()

Returns the message to which the clicked inline button belongs.

id

Returns the query ID. The user clicking the inline button is the one who generated this random ID.

message_id

Returns the message ID to which the clicked inline button belongs.

reply(**args*, ***kwargs*)
Replies to the message (as a reply). Shorthand for *telethon.client.messages.MessageMethods.send_message* with both `entity` and `reply_to` already set.

This method also creates a task to *answer* the callback.

This method will likely fail if *via_inline* is `True`.

respond(**args*, ***kwargs*)
Responds to the message (not as a reply). Shorthand for *telethon.client.messages.MessageMethods.send_message* with `entity` already set.

This method also creates a task to *answer* the callback.

This method will likely fail if *via_inline* is `True`.

via_inline
Whether this callback was generated from an inline button sent via an inline query or not. If the bot sent the message itself with buttons, and one of those is clicked, this will be `False`. If a user sent the message coming from an inline query to the bot, and one of those is clicked, this will be `True`.

If it's `True`, it's likely that the bot is **not** in the chat, so methods like *respond* or *delete* won't work (but *edit* will always work).

classmethod build(*update*)
Builds an event for the given update if possible, or returns None

filter(*event*)
If the ID of `event._chat_peer` isn't in the chats set (or it is but the set is a blacklist) returns `None`, otherwise the event.

The events must have been resolved before this can be called.

class telethon.events.inlinequery.**InlineQuery**(*users=None*, ***, *blacklist_users=False*, *func=None*, *pattern=None*)
Bases: *telethon.events.common.EventBuilder*

Represents an inline query event (when someone writes `'@my_bot query'`).

Args:

users (entity, optional): May be one or more entities (username/peer/etc.), preferably IDs. By default, only inline queries from these users will be handled.

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

pattern (str, callable, Pattern, optional): If set, only queries 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(*query*)
Bases: *telethon.events.common.EventCommon*, *telethon.tl.custom.sendergetter.SenderGetter*

Represents the event of a new callback query.

Members:

query (UpdateBotCallbackQuery): The original UpdateBotCallbackQuery.

Make sure to access the *text* of the query if that's what you want instead working with this.

pattern_match (obj, optional): The resulting object from calling the passed `pattern` function, which is `re.compile(...).match` by default.

answer(*results=None*, *cache_time=0*, ***, *gallery=False*, *next_offset=None*, *private=False*, *switch_pm=None*, *switch_pm_param=''*)

Answers the inline query with the given results.

Args:

results (list, optional): A list of InputBotInlineResult to use. You should use *builder* to create these:

```
builder = inline.builder
r1 = builder.article('Be nice', text='Have a nice day')
r2 = builder.article('Be bad', text="I don't like you")
await inline.answer([r1, r2])
```

You can send up to 50 results as documented in https://core.telegram.org/bots/api#answerinlinequery. Sending more will raise `ResultsTooMuchError`, and you should consider using `next_offset` to paginate them.

cache_time (int, optional): For how long this result should be cached on the user's client. Defaults to 0 for no cache.

gallery (bool, optional): Whether the results should show as a gallery (grid) or not.

next_offset (str, optional): The offset the client will send when the user scrolls the results and it repeats the request.

private (bool, optional): Whether the results should be cached by Telegram (not private) or by the user's client (private).

switch_pm (str, optional): If set, this text will be shown in the results to allow the user to switch to private messages.

switch_pm_param (str, optional): Optional parameter to start the bot with if `switch_pm` was used.

builder

Returns a new *InlineBuilder* instance.

geo

If the user location is requested when using inline mode and the user's device is able to send it, this will return the GeoPoint with the position of the user.

id

Returns the unique identifier for the query ID.

offset

The string the user's client used as an offset for the query. This will either be empty or equal to offsets passed to *answer*.

text

Returns the text the user used to make the inline query.

classmethod build(*update*)

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

filter(*event*)

If the ID of `event._chat_peer` isn't in the chats set (or it is but the set is a blacklist) returns `None`, otherwise the event.

The events must have been resolved before this can be called.

class telethon.events.raw.**Raw**(*types=None*, ***, *func=None*)

Bases: *telethon.events.common.EventBuilder*

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

Args:

types (`list` | `tuple` | `type`, optional): The type or types that the Update instance must be. Equivalent to `if not isinstance(update, types): return`.

classmethod build(*update*)
Builds an event for the given update if possible, or returns None

filter(*event*)
If the ID of `event._chat_peer` isn't in the chats set (or it is but the set is a blacklist) returns `None`, otherwise the event.

The events must have been resolved before this can be called.

resolve(*client*)
Helper method to allow event builders to be resolved before usage

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

telethon.events.**is_handler**(*callback*)
Returns `True` if the given callback is an event handler (i.e. you used *register* on it).

telethon.events.**list**(*callback*)
Returns a list containing the registered event builders inside the specified callback handler.

telethon.events.**register**(*event=None*)
Decorator method to *register* event handlers. This is the client-less *add_event_handler* variant.

Note that this method only registers callbacks as handlers, and does not attach them to any client. This is useful for external modules that don't have access to the client, but still want to define themselves as a handler. Example:

```
>>> from telethon import events
>>> @events.register(events.NewMessage)
... async def handler(event):
...     ...
...
>>> # (somewhere else)
...
>>> from telethon import TelegramClient
```

(continues on next page)

(continued from previous page)

```
>>> client = TelegramClient(...)
>>> client.add_event_handler(handler)
```

Remember that you can use this as a non-decorator through `register(event)(callback)`.

Args:

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

`telethon.events.`**`unregister`**(*callback*, *event=None*)

Inverse operation of *register* (though not a decorator). Client-less *remove_event_handler* variant. **Note that this won't remove handlers from the client**, because it simply can't, so you would generally use this before adding the handlers to the client.

This method is here for symmetry. You will rarely need to unregister events, since you can simply just not add them to any client.

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

telethon.sessions module

telethon.errors package

telethon.errors package

telethon.errors.common module

Errors not related to the Telegram API itself

exception `telethon.errors.common.`**`AlreadyInConversationError`**

Bases: `Exception`

Occurs when another exclusive conversation is opened in the same chat.

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.`**`InvalidBufferError`**(*payload*)

Bases: `BufferError`

Occurs when the buffer is invalid, and may contain an HTTP error code. For instance, 404 means "forgotten/broken authorization key", while

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.`**`MultiError`**

Bases: `Exception`

Exception container for multiple `TLRequest`'s.

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*, *remaining*)
Bases: Exception

Occurs when a type is not found, for example, when trying to read a TLObject with an invalid constructor code.

telethon.errors.rpcbaseerrors module

exception telethon.errors.rpcbaseerrors.**AuthKeyError**(*request*, *message*)
Bases: *telethon.errors.rpcbaseerrors.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.rpcbaseerrors.**BadMessageError**(*request*, *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.rpcbaseerrors.**BadRequestError**(*request*, *message*, *code=None*)
Bases: *telethon.errors.rpcbaseerrors.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.rpcbaseerrors.**BotTimeout**(*request*, *message*)
Bases: *telethon.errors.rpcbaseerrors.RPCError*

Clicking the inline buttons of bots that never (or take to long to) call `answerCallbackQuery` will result in this "special" RPCError.

code = -503

message = 'Timeout'

exception telethon.errors.rpcbaseerrors.**FloodError**(*request*, *message*, *code=None*)
Bases: *telethon.errors.rpcbaseerrors.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.rpcbaseerrors.**ForbiddenError**(*request*, *message*)
Bases: *telethon.errors.rpcbaseerrors.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.rpcbaseerrors.**InvalidDCError**(*request*, *message*, *code=None*)
Bases: *telethon.errors.rpcbaseerrors.RPCError*

The request must be repeated, but directed to a different data center.

code = 303

message = 'ERROR_SEE_OTHER'

exception telethon.errors.rpcbaseerrors.**NotFoundError**(*request*, *message*)
Bases: *telethon.errors.rpcbaseerrors.RPCError*

An attempt to invoke a non-existent object, such as a method.

code = 404

message = 'NOT_FOUND'

exception telethon.errors.rpcbaseerrors.**RPCError**(*request*, *message*, *code=None*)
Bases: Exception

Base class for all Remote Procedure Call errors.

code = None

message = None

exception telethon.errors.rpcbaseerrors.**ServerError**(*request*, *message*)
Bases: *telethon.errors.rpcbaseerrors.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.rpcbaseerrors.**UnauthorizedError**(*request*, *message*, *code=None*)
Bases: *telethon.errors.rpcbaseerrors.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.binaryreader module

This module contains the BinaryReader utility class.

class telethon.extensions.binaryreader.**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.**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.html module

Simple HTML -> Telegram entity parser.

class telethon.extensions.html.**HTMLToTelegramParser**
Bases: html.parser.HTMLParser

handle_data(*text*)

handle_endtag(*tag*)

handle_starttag(*tag*, *attrs*)

telethon.extensions.html.**parse**(*html*)
Parses the given HTML message and returns its stripped representation plus a list of the MessageEntity's that were found.

Parameters **message** – the message with HTML to be parsed.

Returns a tuple consisting of (clean message, [message entities]).

telethon.extensions.html.**unparse**(*text*, *entities*)
Performs the reverse operation to .parse(), effectively returning HTML given a normal text and its MessageEntity's.

Parameters

- **text** – the text to be reconverted into HTML.
- **entities** – the MessageEntity's applied to the text.

Returns a HTML representation of the combination of both inputs.

telethon.network package

telethon.network package

telethon.network.connection module

telethon.network.mtprotoplainsender 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.mtprotoplainsender.**MTProtoPlainSender**(*connection*, ***, *loggers*)

Bases: object

MTProto Mobile Protocol plain sender (https://core.telegram.org/mtproto/description#unencrypted-messages)

send(*request*)

Sends and receives the result for the given request.

telethon.network.mtprotosender module

class telethon.network.mtprotosender.**MTProtoSender**(*auth_key*, *loop*, ***, *loggers*, *retries=5*, *delay=1*, *auto_reconnect=True*, *connect_timeout=None*, *auth_key_callback=None*, *update_callback=None*, *auto_reconnect_callback=None*)

Bases: object

MTProto Mobile Protocol sender (https://core.telegram.org/mtproto/description).

This class is responsible for wrapping requests into `TLMessage`'s, sending them over the network and receiving them in a safe manner.

Automatic reconnection due to temporary network issues is a concern for this class as well, including retry of messages that could not be sent successfully.

A new authorization key will be generated on connection if no other key exists yet.

connect(*connection*)

Connects to the specified given connection using the given auth key.

disconnect()

Cleanly disconnects the instance from the network, cancels all pending requests, and closes the send and receive loops.

disconnected

Future that resolves when the connection to Telegram ends, either by user action or in the background.

Note that it may resolve in either a `ConnectionError` or any other unexpected error that could not be handled.

is_connected()

send(*request*, *ordered=False*)

This method enqueues the given request to be sent. Its send state will be saved until a response arrives, and a `Future` that will be resolved when the response arrives will be returned:

```
async def method():
    # Sending (enqueued for the send loop)
    future = sender.send(request)
    # Receiving (waits for the receive loop to read the result)
    result = await future
```

Designed like this because Telegram may send the response at any point, and it can send other items while one waits for it. Once the response for this future arrives, it is set with the received result, quite similar to how a `receive()` call would otherwise work.

Since the receiving part is "built in" the future, it's impossible to await receive a result that was never sent.

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`**(*sender*)

Executes the authentication process with the Telegram servers.

Parameters **`sender`** – a connected `MTProtoPlainSender`.

Returns returns a (authorization key, time offset) tuple.

`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.tl package

telethon.tl.custom package

telethon.tl.custom package

telethon.tl.custom.draft module

`class` `telethon.tl.custom.draft.`**`Draft`**(*client*, *peer*, *draft*, *entity*)

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).

get_entity()
Returns *entity* but will make an API call if necessary.

get_input_entity()
Returns *input_entity* but will make an API call if necessary.

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=()*)
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=()*, *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".

title (str): Alias for `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(...)`.

is_user (bool): `True` if the `entity` is a User.

is_group (bool): `True` if the `entity` is a Chat or a Channel megagroup.

is_channel (bool): `True` if the `entity` is a Channel.

delete()
Deletes the dialog from your dialog list. If you own the channel this won't destroy it, only delete it from the list.

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.custom.message module

class telethon.tl.custom.message.**Message**(*id*, *to_id=None*, *date=None*, *out=None*, *mentioned=None*, *media_unread=None*, *silent=None*, *post=None*, *from_id=None*, *reply_to_msg_id=None*, *message=None*, *fwd_from=None*, *via_bot_id=None*, *media=None*, *reply_markup=None*, *entities=None*, *views=None*, *edit_date=None*, *post_author=None*, *grouped_id=None*, *from_scheduled=None*, *action=None*)
Bases: *telethon.tl.custom.chatgetter.ChatGetter*, *telethon.tl.custom.sendergetter.SenderGetter*, *telethon.tl.tlobject.TLObject*, abc.ABC

This custom class aggregates both Message and MessageService to ease accessing their members.

Remember that this class implements *ChatGetter* and *SenderGetter* which means you have access to all their sender and chat properties and methods.

Members:

id (int): The ID of this message. This field is *always* present. Any other member is optional and may be `None`.

out (bool): Whether the message is outgoing (i.e. you sent it from another session) or incoming (i.e. someone else sent it).

Note that messages in your own chat are always incoming, but this member will be `True` if you send a message to your own chat. Messages you forward to your chat are *not* considered outgoing, just like official clients display them.

mentioned (bool): Whether you were mentioned in this message or not. Note that replies to your own messages also count as mentions.

media_unread (bool): Whether you have read the media in this message or not, e.g. listened to the voice note media.

silent (bool): Whether this message should notify or not, used in channels.

post (bool): Whether this message is a post in a broadcast channel or not.

from_scheduled (bool): Whether this message was originated from a scheduled one or not.

to_id (Peer): The peer to which this message was sent, which is either PeerUser, PeerChat or PeerChannel. This will always be present except for empty messages.

date (datetime): The UTC+0 `datetime` object indicating when this message was sent. This will always be present except for empty messages.

message (str): The string text of the message for *`Message`* instances, which will be `None` for other types of messages.

action (MessageAction): The message action object of the message for MessageService instances, which will be `None` for other types of messages.

from_id (int): The ID of the user who sent this message. This will be `None` if the message was sent in a broadcast channel.

reply_to_msg_id (int): The ID to which this message is replying to, if any.

fwd_from (MessageFwdHeader): The original forward header if this message is a forward. You should probably use the *`forward`* property instead.

via_bot_id (int): The ID of the bot used to send this message through its inline mode (e.g. "via @like").

media (MessageMedia): The media sent with this message if any (such as photos, videos, documents, gifs, stickers, etc.).

You may want to access the *`photo`*, *`document`* etc. properties instead.

If the media was not present or it was MessageMediaEmpty, this member will instead be `None` for convenience.

reply_markup (ReplyMarkup): The reply markup for this message (which was sent either via a bot or by a bot). You probably want to access *`buttons`* instead.

entities (List[MessageEntity]): The list of markup entities in this message, such as bold, italics, code, hyperlinks, etc.

views (int): The number of views this message from a broadcast channel has. This is also present in forwards.

edit_date (datetime): The date when this message was last edited.

post_author (str): The display name of the message sender to show in messages sent to broadcast channels.

grouped_id (int): If this message belongs to a group of messages (photo albums or video albums), all of them will have the same value here.

action_entities
Returns a list of entities that can took part in this action.

Possible cases for this are MessageActionChatAddUser, types.MessageActionChatCreate, MessageActionChatDeleteUser, MessageActionChatJoinedByLink MessageActionChatMigrateTo and :tl:‘MessageActionChannelMigrateFrom).

If the action is neither of those, the result will be `None`. If some entities could not be retrieved, the list may contain some `None` items in it.

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

button_count
Returns the total button count.

buttons
Returns a matrix (list of lists) containing all buttons of the message as *MessageButton* instances.

click(*i=None*, *j=None*, ***, *text=None*, *filter=None*, *data=None*)
Calls *telethon.tl.custom.messagebutton.MessageButton.click* for the specified button.

Does nothing if the message has no buttons.

Args:

i (int): Clicks the i’th button (starting from the index 0). Will `raise IndexError` if out of bounds. Example:

```
>>> message = ...  # get the message somehow
>>> # Clicking the 3rd button
>>> # [button1] [button2]
>>> # [     button3     ]
>>> # [button4] [button5]
>>> message.click(2)  # index
```

j (int): Clicks the button at position (i, j), these being the indices for the (row, column) respectively. Example:

```
>>> # Clicking the 2nd button on the 1st row.
>>> # [button1] [button2]
>>> # [     button3     ]
>>> # [button4] [button5]
>>> message.click(0, 1)  # (row, column)
```

This is equivalent to `message.buttons[0][1].click()`.

text (str | callable): Clicks the first button with the text “text”. This may also be a callable, like a `re.compile(...).match`, and the text will be passed to it.

filter (callable): Clicks the first button for which the callable returns `True`. The callable should accept a single *telethon.tl.custom.messagebutton.MessageButton* argument.

data (bytes): This argument overrides the rest and will not search any buttons. Instead, it will directly send the request to behave as if it clicked a button with said data. Note that if the message does not have this data, it will `raise DataInvalidError`.

client
Returns the `telethon.client.telegramclient.TelegramClient` that patched this message. This will only be present if you **use the friendly methods**, it won't be there if you invoke raw API methods manually, in which case you should only access members, not properties.

contact
If the message media is a contact, this returns the MessageMediaContact.

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. Shorthand for *telethon.client.messages.MessageMethods.delete_messages* with `entity` and `message_ids` already set.

If you need to delete more than one message at once, don't use this *delete* method. Use a `telethon.client.telegramclient.TelegramClient` instance directly.

document
If the message media is a document, this returns the Document object.

download_media(**args*, ***kwargs*)
Downloads the media contained in the message, if any. Shorthand for *telethon.client.downloads.DownloadMethods.download_media* with the `message` already set.

edit(**args*, ***kwargs*)
Edits the message iff it's outgoing. Shorthand for *telethon.client.messages.MessageMethods.edit_message* with both `entity` and `message` already set.

Returns `None` if the message was incoming, or the edited *Message* otherwise.

Note: This is different from *client.edit_message* and **will respect** the previous state of the message. For example, if the message didn't have a link preview, the edit won't add one by default, and you should force it by setting it to `True` if you want it.

This is generally the most desired and convenient behaviour, and will work for link previews and message buttons.

---

forward
Returns *Forward* if the message has been forwarded from somewhere else.

forward_to(**args*, ***kwargs*)
Forwards the message. Shorthand for *telethon.client.messages.MessageMethods.forward_messages* with both `messages` and `from_peer` already set.

If you need to forward more than one message at once, don't use this *forward_to* method. Use a `telethon.client.telegramclient.TelegramClient` instance directly.

game
If the message media is a game, this returns the Game.

geo
If the message media is geo, geo live or a venue, this returns the GeoPoint.

get_buttons()
Returns *buttons*, but will make an API call to find the input chat (needed for the buttons) unless it's already cached.

get_entities_text(*cls=None*)
Returns a list of tuples [(MessageEntity, `str`)], the string being the inner text of the message entity (like bold, italics, etc).

Args:

cls (type): Returns entities matching this type only. For example, the following will print the text for all `code` entities:

```
>>> from telethon.tl.types import MessageEntityCode
>>>
>>> m = ...  # get the message
>>> for _, inner_text in m.get_entities_text(MessageEntityCode):
>>>     print(inner_text)
```

get_reply_message()
The *Message* that this message is replying to, or `None`.

The result will be cached after its first use.

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

invoice
If the message media is an invoice, this returns the MessageMediaInvoice.

is_reply
True if the message is a reply to some other.

Remember that you can access the ID of the message this one is replying to through `reply_to_msg_id`, and the *Message* object with *get_reply_message()*.

photo
If the message media is a photo, this returns the Photo object. This will also return the photo for MessageService if their action is MessageActionChatEditPhoto.

poll
If the message media is a poll, this returns the MessageMediaPoll.

raw_text
The raw message text, ignoring any formatting. Will be `None` for MessageService.

Setting a value to this field will erase the `entities`, unlike changing the `message` member.

reply(**args*, ***kwargs*)
Replies to the message (as a reply). Shorthand for *telethon.client.messages.MessageMethods.send_message* with both `entity` and `reply_to` already set.

respond(**args*, ***kwargs*)
Responds to the message (not as a reply). Shorthand for *telethon.client.messages.MessageMethods.send_message* with `entity` already set.

sticker
If the message media is a document with a Sticker attribute, this returns the Document object.

text
The message text, formatted using the client's default parse mode. Will be `None` for MessageService.

venue
If the message media is a venue, this returns the MessageMediaVenue.

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.

web_preview
If the message has a loaded web preview, this returns the WebPage object.

telethon.tl.custom.messagebutton module

class telethon.tl.custom.messagebutton.**MessageButton**(*client*, *original*, *chat*, *bot*, *msg_id*)
Bases: object

Note: *Message.buttons* are instances of this type. If you want to **define** a reply markup for e.g. sending messages, refer to *Button* instead.

Custom class that encapsulates a message button providing an abstraction to easily access some commonly needed features (such as clicking the button itself).

Attributes:

button (KeyboardButton): The original KeyboardButton object.

click()
Emulates the behaviour of clicking this button.

If it's a normal KeyboardButton with text, a message will be sent, and the sent *telethon.tl.custom.message.Message* returned.

If it's an inline KeyboardButtonCallback with text and data, it will be "clicked" and the BotCallbackAnswer returned.

If it's an inline KeyboardButtonSwitchInline button, the StartBotRequest will be invoked and the resulting updates returned.

If it's a KeyboardButtonUrl, the URL of the button will be passed to webbrowser.open and return True on success.

client
Returns the telethon.client.telegramclient.TelegramClient instance that created this instance.

data
The bytes data for KeyboardButtonCallback objects.

inline_query
The query str for KeyboardButtonSwitchInline objects.

text
The text string of the button.

url
The url str for KeyboardButtonUrl objects.

telethon.tl.custom.forward module

class telethon.tl.custom.forward.**Forward**(*client*, *original*, *entities*)
Bases: *telethon.tl.custom.chatgetter.ChatGetter*, *telethon.tl.custom.sendergetter.SenderGetter*

Custom class that encapsulates a MessageFwdHeader providing an abstraction to easily access information like the original sender.

Remember that this class implements *ChatGetter* and *SenderGetter* which means you have access to all their sender and chat properties and methods.

Attributes:

original_fwd (MessageFwdHeader): The original MessageFwdHeader instance.

Any other attribute: Attributes not described here are the same as those available in the original MessageFwdHeader.

telethon.tl.custom.button module

class telethon.tl.custom.button.**Button**(*button*, *, *resize*, *single_use*, *selective*)
Bases: object

Note: This class is used to **define** reply markups, e.g. when sending a message or replying to events. When you access *Message.buttons* they are actually *MessageButton*, so you might want to refer to that class instead.

Helper class to allow defining `reply_markup` when sending a message with inline or keyboard buttons.

You should make use of the defined class methods to create button instances instead making them yourself (i.e. don't do `Button(...)` but instead use methods line *Button.inline(...)* etc.

You can use *inline*, *switch_inline* and *url* together to create inline buttons (under the message).

You can use *text*, *request_location* and *request_phone* together to create a reply markup (replaces the user keyboard). You can also configure the aspect of the reply with these.

You **cannot** mix the two type of buttons together, and it will error if you try to do so.

The text for all buttons may be at most 142 characters. If more characters are given, Telegram will cut the text to 128 characters and add the ellipsis (…) character as the 129.

static clear()
Clears all the buttons. When used, no other button should be present or it will be ignored.

static force_reply()
Forces a reply. If used, no other button should be present or it will be ignored.

static inline(*text*, *data=None*)
Creates a new inline button.

If `data` is omitted, the given *text* will be used as `data`. In any case `data` should be either `bytes` or `str`.

Note that the given `data` must be less or equal to 64 bytes. If more than 64 bytes are passed as data, `ValueError` is raised.

classmethod request_location(*text*, *, *resize=None*, *single_use=None*, *selective=None*)
Creates a new button that will request the user's location upon being clicked.

`resize`, `single_use` and `selective` are documented in *text*.

classmethod request_phone(*text*, *, *resize=None*, *single_use=None*, *selective=None*)
Creates a new button that will request the user's phone number upon being clicked.

`resize`, `single_use` and `selective` are documented in *text*.

static switch_inline(*text*, *query=''*, *same_peer=False*)
Creates a new button to switch to inline query.

If `query` is given, it will be the default text to be used when making the inline query.

If `same_peer is True` the inline query will directly be set under the currently opened chat. Otherwise, the user will have to select a different dialog to make the query.

classmethod text(*text*, ***, *resize=None*, *single_use=None*, *selective=None*)
Creates a new button with the given text.

Args:

resize (bool): If present, the entire keyboard will be reconfigured to be resized and be smaller if there are not many buttons.

single_use (bool): If present, the entire keyboard will be reconfigured to be usable only once before it hides itself.

selective (bool): If present, the entire keyboard will be reconfigured to be "selective". The keyboard will be shown only to specific users. It will target users that are @mentioned in the text of the message or to the sender of the message you reply to.

static url(*text*, *url=None*)
Creates a new button to open the desired URL upon clicking it.

If no *url* is given, the *text* will be used as said URL instead.

telethon.tl.custom.inlinebuilder module

class telethon.tl.custom.inlinebuilder.**InlineBuilder**(*client*)
Bases: object

Helper class to allow defining *InlineQuery* results.

Common arguments to all methods are explained here to avoid repetition:

text (str, optional): If present, the user will send a text message with this text upon being clicked.

link_preview (bool, optional): Whether to show a link preview in the sent text message or not.

geo (InputGeoPoint, GeoPoint, InputMediaVenue, MessageMediaVenue, optional): If present, it may either be a geo point or a venue.

period (int, optional): The period in seconds to be used for geo points.

contact (InputMediaContact, MessageMediaContact, optional): If present, it must be the contact information to send.

game (bool, optional): May be `True` to indicate that the game will be sent.

buttons (list, *custom.Button*, KeyboardButton, optional): Same as `buttons` for *client.send_message*.

parse_mode (str, optional): Same as `parse_mode` for *client.send_message*.

id (str, optional): The string ID to use for this result. If not present, it will be the SHA256 hexadecimal digest of converting the created InputBotInlineResult with empty ID to `bytes()`, so that the ID will be deterministic for the same input.

> **Note:** If two inputs are exactly the same, their IDs will be the same too. If you send two articles with the same ID, it will raise `ResultIdDuplicateError`. Consider giving them an explicit ID if you need to send two results that are the same.

article(*title*, *description=None*, ***, *url=None*, *thumb=None*, *content=None*, *id=None*, *text=None*, *parse_mode=()*, *link_preview=True*, *geo=None*, *period=60*, *contact=None*, *game=False*, *buttons=None*)
Creates new inline result of article type.

Args:

title (`str`): The title to be shown for this result.

description (`str`, optional): Further explanation of what this result means.

url (`str`, optional): The URL to be shown for this result.

thumb (InputWebDocument, optional): The thumbnail to be shown for this result. For now it has to be a InputWebDocument if present.

content (InputWebDocument, optional): The content to be shown for this result. For now it has to be a InputWebDocument if present.

document(*file*, *title=None*, ***, *description=None*, *type=None*, *mime_type=None*, *attributes=None*, *force_document=False*, *voice_note=False*, *video_note=False*, *use_cache=True*, *id=None*, *text=None*, *parse_mode=()*, *link_preview=True*, *geo=None*, *period=60*, *contact=None*, *game=False*, *buttons=None*)
Creates a new inline result of document type.

`use_cache`, `mime_type`, `attributes`, `force_document`, `voice_note` and `video_note` are described in *`client.send_file`*.

Args:

file (`obj`): Same as `file` for *`client.send_file`*.

title (`str`, optional): The title to be shown for this result.

description (`str`, optional): Further explanation of what this result means.

type (`str`, optional): The type of the document. May be one of: photo, gif, mpeg4_gif, video, audio, voice, document, sticker.

See "Type of the result" in https://core.telegram.org/bots/api.

game(*short_name*, ***, *id=None*, *text=None*, *parse_mode=()*, *link_preview=True*, *geo=None*, *period=60*, *contact=None*, *game=False*, *buttons=None*)
Creates a new inline result of game type.

Args:

short_name (`str`): The short name of the game to use.

photo(*file*, ***, *id=None*, *text=None*, *parse_mode=()*, *link_preview=True*, *geo=None*, *period=60*, *contact=None*, *game=False*, *buttons=None*)
Creates a new inline result of photo type.

Args:

file (`obj`, optional): Same as `file` for *`client.send_file`*.

telethon.tl.custom.inlineresult module

class telethon.tl.custom.inlineresult.**InlineResult**(*client*, *original*, *query_id=None*)
Bases: object

Custom class that encapsulates a bot inline result providing an abstraction to easily access some commonly needed features (such as clicking a result to select it).

Attributes:

result (BotInlineResult): The original BotInlineResult object.

ARTICLE = 'article'

AUDIO = 'audio'

CONTACT = 'contact'

DOCUMENT = 'document'

GAME = 'game'

GIF = 'gif'

LOCATION = 'location'

PHOTO = 'photo'

VENUE = 'venue'

VIDEO = 'video'

VIDEO_GIF = 'mpeg4_gif'

click(*entity*, *reply_to=None*, *silent=False*, *clear_draft=False*, *hide_via=False*)
Clicks this result and sends the associated *message*.

Args:

entity (entity): The entity to which the message of this result should be sent.

reply_to (int | *Message*, optional): If present, the sent message will reply to this ID or message.

silent (bool, optional): If True, the sent message will not notify the user(s).

clear_draft (bool, optional): Whether the draft should be removed after sending the message from this result or not. Defaults to False.

hide_via (bool, optional): Whether the "via @bot" should be hidden or not. Only works with certain bots (like @bing or @gif).

description
The description for this inline result. It may be None.

document
Returns either the WebDocument content for normal results or the Document for media results.

download_media(**args*, ***kwargs*)
Downloads the media in this result (if there is a document, the document will be downloaded; otherwise, the photo will if present).

This is a wrapper around *client.download_media*.

message
The always-present BotInlineMessage that will be sent if *click* is called on this result.

photo
Returns either the WebDocument thumbnail for normal results or the Photo for media results.

title
The title for this inline result. It may be `None`.

type
The always-present type of this result. It will be one of: `'article'`, `'photo'`, `'gif'`, `'mpeg4_gif'`, `'video'`, `'audio'`, `'voice'`, `'document'`, `'location'`, `'venue'`, `'contact'`, `'game'`.

You can access all of these constants through *InlineResult*, such as *InlineResult.ARTICLE*, *InlineResult.VIDEO_GIF*, etc.

url
The URL present in this inline results. If you want to "click" this URL to open it in your browser, you should use Python's `webbrowser.open(url)` for such task.

telethon.tl.custom.inlineresults module

class telethon.tl.custom.inlineresults.**InlineResults**(*client*, *original*)
Bases: `list`

Custom class that encapsulates BotResults providing an abstraction to easily access some commonly needed features (such as clicking one of the results to select it)

Note that this is a list of *InlineResult* so you can iterate over it or use indices to access its elements. In addition, it has some attributes.

Attributes:

result (BotResults): The original BotResults object.

query_id (int): The random ID that identifies this query.

cache_time (int): For how long the results should be considered valid. You can call *results_valid* at any moment to determine if the results are still valid or not.

users (User): The users present in this inline query.

gallery (bool): Whether these results should be presented in a grid (as a gallery of images) or not.

next_offset (str, optional): The string to be used as an offset to get the next chunk of results, if any.

switch_pm (InlineBotSwitchPM, optional): If presents, the results should show a button to switch to a private conversation with the bot using the text in this object.

results_valid()
Returns `True` if the cache time has not expired yet and the results can still be considered valid.

telethon.tl.custom.chatgetter module

class telethon.tl.custom.chatgetter.**ChatGetter**
Bases: `abc.ABC`

Helper base class that introduces the *chat*, *input_chat* and *chat_id* properties and *get_chat* and *get_input_chat* methods.

Subclasses **must** have the following private members: `_chat`, `_input_chat`, `_chat_peer`, `_broadcast` and `_client`. As an end user, you should not worry about this.

chat
Returns the User, Chat or Channel where this object belongs to. It may be `None` if Telegram didn't send the chat.

If you're using *telethon.events*, use *get_chat* instead.

chat_id
Returns the marked chat integer ID. Note that this value **will be different** from `to_id` for incoming private messages, since the chat *to* which the messages go is to your own person, but the *chat* itself is with the one who sent the message.

TL;DR; this gets the ID that you expect.

get_chat()
Returns *chat*, but will make an API call to find the chat unless it's already cached.

get_input_chat()
Returns *input_chat*, but will make an API call to find the input chat unless it's already cached.

input_chat
This InputPeer is the input version of the chat where the message was sent. Similarly to `input_sender`, 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 doesn't have enough information available.

is_channel
True if the message was sent on a megagroup or channel.

is_group
True if the message was sent on a group or megagroup.

is_private
True if the message was sent as a private message.

telethon.tl.custom.sendergetter module

class telethon.tl.custom.sendergetter.**SenderGetter**
Bases: `abc.ABC`

Helper base class that introduces the *sender*, *input_sender* and *sender_id* properties and *get_sender* and *get_input_sender* methods.

Subclasses **must** have the following private members: `_sender`, `_input_sender`, `_sender_id` and `_client`. As an end user, you should not worry about this.

get_input_sender()
Returns *input_sender*, but will make an API call to find the input sender unless it's already cached.

get_sender()
Returns *sender*, but will make an API call to find the sender unless it's already cached.

input_sender
This InputPeer is the input version of the user/channel 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.

sender
Returns the User or Channel that sent this object. It may be `None` if Telegram didn't send the sender.

If you're using *telethon.events*, use *get_sender* instead.

sender_id
Returns the marked sender integer ID, if present.

telethon.tl.custom.conversation module

class telethon.tl.custom.conversation.**Conversation**(*client*, *input_chat*, *, *timeout*, *total_timeout*, *max_messages*, *exclusive*, *replies_are_responses*)

Bases: *telethon.tl.custom.chatgetter.ChatGetter*

Represents a conversation inside an specific chat.

A conversation keeps track of new messages since it was created until its exit and easily lets you query the current state.

If you need a conversation across two or more chats, you should use two conversations and synchronize them as you better see fit.

cancel()
Cancels the current conversation and exits the context manager.

get_edit(*message=None*, *, *timeout=None*)
Awaits for an edit after the last message to arrive. The arguments are the same as those for *get_response*.

get_reply(*message=None*, *, *timeout=None*)
Returns a coroutine that will resolve once a reply (that is, a message being a reply) arrives. The arguments are the same as those for *get_response*.

get_response(*message=None*, *, *timeout=None*)
Returns a coroutine that will resolve once a response arrives.

Args:

message (*Message* | int, optional): The message (or the message ID) for which a response is expected. By default this is the last sent message.

timeout (int | float, optional): If present, this `timeout` (in seconds) will override the per-action timeout defined for the conversation.

mark_read(*message=None*)
Marks as read the latest received message if `message is None`. Otherwise, marks as read until the given message (or message ID).

This is equivalent to calling *client.send_read_acknowledge*.

send_file(**args*, ***kwargs*)
Sends a file in the context of this conversation. Shorthand for *telethon.client.uploads.UploadMethods.send_file* with `entity` already set.

send_message(**args*, ***kwargs*)
Sends a message in the context of this conversation. Shorthand for *telethon.client.messages.MessageMethods.send_message* with `entity` already set.

wait_event(*event*, *, *timeout=None*)
Waits for a custom event to occur. Timeouts still apply.

Unless you're certain that your code will run fast enough, generally you should get a "handle" of this special coroutine before acting. Generally, you should do this:

```
>>> from telethon import TelegramClient, events
>>>
>>> client = TelegramClient(...)
>>>
>>> async def main():
>>>     async with client.conversation(...) as conv:
>>>         response = conv.wait_event(events.NewMessage(incoming=True))
>>>         await conv.send_message('Hi')
>>>         response = await response
```

This way your event can be registered before acting, since the response may arrive before your event was registered. It depends on your use case since this also means the event can arrive before you send a previous action.

wait_read(*message=None*, *, *timeout=None*)
Awaits for the sent message to be read. Note that receiving a response doesn't imply the message was read, and this action will also trigger even without a response.

telethon.tl.custom.adminlogevent module

class telethon.tl.custom.adminlogevent.**AdminLogEvent**(*original*, *entities*)
Bases: object

Represents a more friendly interface for admin log events.

Members:

original (ChannelAdminLogEvent): The original ChannelAdminLogEvent.

entities (dict): A dictionary mapping user IDs to User.

When *old* and *new* are ChannelParticipant, you can use this dictionary to map the user_id, kicked_by, inviter_id and promoted_by IDs to their User.

user (User): The user that caused this action (entities[original.user_id]).

input_user (InputPeerUser): Input variant of user.

action
The original ChannelAdminLogEventAction.

changed_about
Whether the channel's about was changed in this event or not.

If True, *old* and *new* will be present as str.

changed_admin
Whether the permissions for an admin in this channel changed in this event or not.

If True, *old* and *new* will be present as ChannelParticipant.

changed_default_banned_rights
Whether the default banned rights were changed in this event or not.

If True, *old* and *new* will be present as ChatBannedRights.

changed_hide_history
Whether hiding the previous message history for new members in the channel were toggled in this event or not.

If True, *old* and *new* will be present as bool.

changed_invites
Whether the invites in the channel were toggled in this event or not.

If `True`, *old* and *new* will be present as `bool`.

changed_message
Whether a message in this channel was edited in this event or not.

If `True`, *old* and *new* will be present as *Message*.

changed_photo
Whether the channel's photo was changed in this event or not.

If `True`, *old* and *new* will be present as ChatPhoto.

changed_pin
Whether a new message in this channel was pinned in this event or not.

If `True`, *new* will be present as *Message*.

changed_restrictions
Whether a message in this channel was edited in this event or not.

If `True`, *old* and *new* will be present as ChannelParticipant.

changed_signatures
Whether the message signatures in the channel were toggled in this event or not.

If `True`, *old* and *new* will be present as `bool`.

changed_sticker_set
Whether the channel's sticker set was changed in this event or not.

If `True`, *old* and *new* will be present as InputStickerSet.

changed_title
Whether the channel's title was changed in this event or not.

If `True`, *old* and *new* will be present as `str`.

changed_username
Whether the channel's username was changed in this event or not.

If `True`, *old* and *new* will be present as `str`.

date
The date when this event occured.

deleted_message
Whether a message in this channel was deleted in this event or not.

If `True`, *old* will be present as *Message*.

id
The ID of this event.

joined
Whether `user` joined through the channel's public username in this event or not.

joined_invite
Whether a new user joined through an invite link to the channel in this event or not.

If `True`, *new* will be present as ChannelParticipant.

left
Whether `user` left the channel in this event or not.

new
The new value present in the event.

old
The old value from the event.

stopped_poll
Whether a poll was stopped in this event or not.

If `True`, *new* will be present as *Message*.

stringify()

user_id
The ID of the user that triggered this event.

telethon.tl.tlobject module

class telethon.tl.tlobject.**TLObject**
Bases: object

CONSTRUCTOR_ID = None

SUBCLASS_OF_ID = None

classmethod from_reader(*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.

static serialize_bytes(*data*)
Write bytes by using Telegram guidelines

static serialize_datetime(*dt*)

stringify()

to_dict()

to_json(*fp=None*, *default=<function _json_default>*, ***kwargs*)
Represent the current *TLObject* as JSON.

If `fp` is given, the JSON will be dumped to said file pointer, otherwise a JSON string will be returned.

Note that bytes and datetimes cannot be represented in JSON, so if those are found, they will be base64 encoded and ISO-formatted, respectively, by default.

class telethon.tl.tlobject.**TLRequest**
Bases: *telethon.tl.tlobject.TLObject*

Represents a content-related *TLObject* (a request that can be sent).

static read_result(*reader*)

resolve(*client*, *utils*)

Module contents

class telethon.**TelegramClient**(*session*, *api_id*, *api_hash*, *, *connection=<class 'telethon.network.connection.tcpfull.ConnectionTcpFull'>*, *use_ipv6=False*, *proxy=None*, *timeout=10*, *request_retries=5*, *connection_retries=5*, *retry_delay=1*, *auto_reconnect=True*, *sequential_updates=False*, *flood_sleep_threshold=60*, *device_model=None*, *system_version=None*, *app_version=None*, *lang_code='en'*, *system_lang_code='en'*, *loop=None*, *base_logger=None*)

Bases: *telethon.client.account.AccountMethods*, *telethon.client.auth.AuthMethods*, *telethon.client.downloads.DownloadMethods*, *telethon.client.dialogs.DialogMethods*, *telethon.client.chats.ChatMethods*, *telethon.client.bots.BotMethods*, *telethon.client.messages.MessageMethods*, *telethon.client.uploads.UploadMethods*, *telethon.client.buttons.ButtonMethods*, *telethon.client.updates.UpdateMethods*, *telethon.client.messageparse.MessageParseMethods*, *telethon.client.users.UserMethods*

1.31 Indices and tables

- genindex
- modindex
- search

Python Module Index

t

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W